REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-

AFRL-SR-BL-TR-98-

0408

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TY
	1993	Final

4. TITLE AND SUBTITLE
Application of a Genetic Algorithm to the Optimization of a Missile Autopilot Controller for Performance Criteria with Non-Analytic Solutions

5. FUNDING NUMBERS

6. AUTHORS
Richard A. Hull

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Central Florida

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NI
110 Duncan Avenue, Room B-115
Bolling Air Force Base, DC 20332-8080

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
See attached.

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Abstract

Modern optimal control theory provides analytic solutions for a set of linear feedback design problems with linear quadratic performance criteria. Recent progress in the field of robust multivariable feedback design has incorporated additional constraints which have addressed the classical concerns with stability margins, system sensitivity and disturbance rejection. Despite these important advances, many practical design problems arise in which the desired system performance constraints cannot be accommodated by the available theoretic techniques.

Genetic algorithms (GA's), on the other hand, offer a numerical search method which does not require a statement of the mathematical relationship between the performance criteria and the parameter update rule. The objective of this thesis is to demonstrate that GA's provide a method of optimizing control system problems with analytically intractable constraints.

A linear missile airframe and actuator state space model is developed with linear feedback controller, and implemented in a discrete time simulation. A genetic algorithm is constructed to optimize the linear controller parameters, first with respect to a weighted linear quadratic performace index. Additional performance constraints are then imposed to meet rise time, peak actuator effort, and settling error specifications. Computer simulation results show that the genetic algorithm provides good convergence to near optimal controller designs for each successive combination of constraints.

Application of a Genetic Algorithm to the Optimization of a Missile Autopilot Controller for Performance Criteria with Non-Analytic Solutions

by

Richard A. Hull
B.S. E.S.M., University of Florida, 1972

THESIS

Submitted in partial fulfillment of the requirements
for the degree of
Master of Science in Electrical Engineering
College of Engineering
University of Central Florida
Orlando, Florida

Spring Term
1993

Abstract

Modern optimal control theory provides analytic solutions for a set of linear feedback design problems with linear quadratic performance criteria. Recent progress in the field of robust multivariable feedback design has incorporated additional constraints which have addressed the classical concerns with stability margins, system sensitivity and disturbance rejection. Despite these important advances, many practical design problems arise in which the desired system performance constraints cannot be accommodated by the available theoretic techniques.

Genetic algorithms (GA's), on the other hand, offer a numerical search method which does not require a statement of the mathematical relationship between the performance criteria and the parameter update rule. The objective of this thesis is to demonstrate that GA's provide a method of optimizing control system problems with analytically intractable constraints.

A linear missile airframe and actuator state space model is developed with linear feedback controller, and implemented in a discrete time simulation. A genetic algorithm is constructed to optimize the linear controller parameters, first with respect to a weighted linear quadratic performace index. Additional performance constraints are then imposed to meet rise time, peak actuator effort, and settling error specifications. Computer simulation results show that the genetic algorithm provides good convergence to near optimal controller designs for each successive combination of constraints.

Contents

List of Figures

CHAPTER 1

INTRODUCTION

Classical autopilot design methods rely on frequency domain techniques to achieve gain and phase margin criteria, while providing desired characteristics of the closed loop system response such as minimum rise time and maximum overshoot.

Autopilot design methods based on modern state space concepts have been developed which allow the designer to shape the dynamic response of the closed loop system by placing the poles of the compensated system, provided issues of *controllability* and *observabibility* have been adaquately addressed. Determining where to place the poles, however, is somewhat more subjective.

Recent progress in the field of robust multivariable feedback design theory has employed singular value decomposition as a measure of system performance and robustness. Singular value loop shaping has then been accomplished via H_∞ , frequency weighted LQG, and LQG loop transfer recovery design techniques [11,4].

All of these important advances, however, have placed constraints on the ways in which system performance and robustness characteristics must be specified. Most control system engineers will be faced with many design problems in which the given specifications cannot readily be formulated in terms required by the available theoretic design criteria. In particular, many long accepted norms of classical system performance criteria, such as rise time and maximum overshoot, may result in

problem formulations which do not have analytic solutions.

Genetic algorithms (GA's), on the other hand, offer a numerical search method which mathematically mimics some of the aspects of biological natural selection in order to achieve adaptation to a specific measure of fitness. An important advantage of genetic algorithms is that they do not require a mathematical statement of the relationship between the performance criteria and the parameter update rule. Instead they rely on reproductive characteristics of a sample gene pool which is bred to maximize the desired fitness measure. Furrthermore, genetic algorithms are good at finding *global* as opposed to *local* optimums. These properties make GA's an excellent candidate for numerically optimizing a multicriterion design specification problem which incorporates an arbitrary but feasible combination of performance and constraint criteria.

The objective of this thesis is to demonstrate that a genetic algorithm provides a method for optimizing an autopilot control system with respect to a set of design specifications which are feasible but have an analytically intractable solution.

1.1 The Control System Optimization Problem

The controller design problem begins with a model of the system to be controlled, referred to as the *plant* , and a set of design goals, or *specifications* . The plant in this instance will be regarded as the physical system (i.e. missile airframe) along with its control surface actuators and feedback sensors. The design specifications must incorporate criteria related to the performance, stability, and robustness of the resulting controlled system.

In the general case, the form of the controller, as well as the form of the design specifications, may all be open for selection by the designer. In fact, in

the most general case, decisions regarding the design of the plant, the number and fidelity of feedback sensors, the size and location of control surfaces, the bandwidth of actuators, and the like, may also be determined or influenced by the control systems engineer. At some stage in the design process, however, the other factors become fixed, and the controller design problem becomes a search for a suitable controller that will satisfy the stated design specifications.

In optimal control theory, the goal of finding the *best* controller is introduced. To this goal, the design specifications are formulated in mathematical terms as a performance measure plus auxillary constraints. The objective of the optimal control problem is to determine the controller that will minimize (or maximize) the performance measure, and at the same time satisfy the auxillary constraints [9].

In practice, most optimal control procedures fix the form of the controller and codify the form of the performance measure in order to develop analytical solutions. Most optimal controls work is based on Linear Time Invariant Causal (LTIC) Systems with linear state feedback controllers. The best known of these methods employs the weighted mean-square error of the regulation variables and control effort to form a quadratic performance index. This usually results in the Linear Quadratic Regulator (LQR) problem when full state feedback is available, or the Linear Quadratic Gaussian (LQG) problem when state estimation is required. The LQR design, while extremely robust from a theoretical standpoint, is often impractical to implement, and the LQG design results in a controller that is notoriously non-robust.

In their book **Linear Controller Design, Limits of Performance** , Boyd and Barratt [2] develop an analytical framework for determining the most stringent set of controller design specifications, within a specific class, that can be met using any controller design method for a given plant and feedback control configura-

tion. This results in the so-called *Pareto optimal specification* , and the resulting controller is the *Pareto optimal design* . After classifying many types of design specifications, Boyd and Barratt present a numerical method for solving controller design problems with closed-loop convex nondifferentiable design specifications that do not have analytic solutions. While this is an impressive and inspiring body of work, such *Pareto* optimal designs usually result in controllers of inordinatly high order (say 100 or more) for even relatively simple systems !

For the purpose of this thesis, *the control system optimization problem* will be limited to a fixed form of controller: linear state feedback control; and a given Linear Time Invariant Causal (LTIC) plant. A performace measure and several auxillary constraints will be successively incorporated into a single *fitness measure* that can be maximized by a genetic algorithm. It will be demonstrated that the genetic algorithm can be successfully applied to the resulting multicriterion optimization problems for which analytic solutions do not exist.

1.2 Common Measures of Performance

Control system design specificatons must incorporate criteria related to the performance, stability, and robustness of the resulting controlled system. In classical controls design methods, performance specifications describe how the closed loop system should respond to commanded inputs and how well it should reject uncommanded disturbances. Some examples of classical performance specifications include: rise time, percent overshoot, and settling time [14].

Robustness specifications are intended to limit the variations in performance of the closed loop system due to variations in the plant during operation, or due to errors and simplifications in the plant model used for design. In classical de-

sign methods, robustness specifications usually take the form of allowable gain and phase margins. Closed loop system stability is usually understood to be a *hard* constraint that is contained within the performance specifications and guaranteed by the robustness specifications.

Modern optimal control theory provides an analytical solution to the Linear Quadratic Regulator (LQR) problem by minimizing a *quadratic performance index* of the form :

$$J = \int_0^T [\mathbf{x}'(t)\mathbf{Q}\mathbf{x}(t) + \mathbf{u}'(t)\mathbf{R}\mathbf{u}(t)]\, dt \tag{1.1}$$

where :

$\mathbf{x}(t)$	is the state variable vector,
$\mathbf{u}(t)$	is the control variable vector,
$\mathbf{Q}$	is a positive semi-definite weighting matrix
$\mathbf{R}$	is a positive definite weighting matrix
T	is the finite final time

For the dynamic process characterized by :

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \tag{1.2}$$

solution of the Algebraic Riccati Equation (ARE) will provide a steady state linear control law of the form :

$$\mathbf{u}(t) = -\mathbf{k}\mathbf{x}(t) \tag{1.3}$$

where $\mathbf{k}$ is the steady state gain matrix that minimizes the quadratic performance index J. This basic perfomance index can be modified to incorporate numerous variations such as:

1. final value of state vector at time T

2. time varying **Q** matrix

3. time varying **R** matrix

4. non-zero initial time

5. infinite final time ($T = \infty$)

1.3 Standard Optimization Techniques

The solution of the control system optimization problem depends to a large extent upon certain properties of the design specification. In a few cases closed form *analytical* solutions may exist, but usually a form of numerical search technique is required. Many forms of search techniques have been documented, but most fall into one of three broad categories: *calculus-based, enumerative, or random* [7].

If the design specification is *differentiable* , then several standard analytical as well as calculus-based search techniques may apply. If the design specification is not differentiable, but is *closed-loop convex* [2, chapter 6] then numerical search methods such as the *Nelder-Meade algorithm* [12], the *cutting plane algorithm* [2], or the *ellipsoid algorithm* [2] may be required. If the design specification is non-differentiable and non-convex, then the more time consuming enumerative or random search techniques may be the only hope.

The quadratic performance index shown in equation 1.1 is a differentiable specification, and the Linear Quadratic Regulator problem has been analytically solved in its many forms by application of the Hamilton-Jacobi equation as developed in either [1, chapter 2] or [9, chapter 5, section 2]. The solution of the resulting two-point boundary value problem results in an optimal control of the form:

$$\mathbf{u}^*(t) = -\mathbf{k}(t)\mathbf{x}(t) = -\mathbf{R}^{-1}(t)\mathbf{B}'(t)\mathbf{P}(t)\mathbf{x}(t) \tag{1.4}$$

where $\mathbf{P}(t)$ is the solution to the familiar Riccati equation:

$$\dot{\mathbf{P}}(t) = -\mathbf{P}(t)\mathbf{A}(t) - \mathbf{A}'(t)\mathbf{P}(t) - \mathbf{Q}(t) + \mathbf{P}(t)\mathbf{B}(t)\mathbf{R}^{-1}(t)\mathbf{B}'(t)\mathbf{P}(t) \qquad (1.5)$$

The optimal LQR control is thus time varying, and the Riccati equation can be solved analytically for simple systems, or numerically for more complicated systems by integrating backward from the final time T. Methods of solving the Riccati equation are presented in many standard texts, see [9,1,5,6,3] for details.

The steady state solution of equation 1.5 can be determined by setting the derivative to zero and solving the resulting Algebraic Riccati Equation (ARE):

$$0 = -\mathbf{PA} - \mathbf{A}'\mathbf{P} - \mathbf{Q} + \mathbf{PBR}^{-1}\mathbf{B}'\mathbf{P} \qquad (1.6)$$

It has been shown that if the system is asymptotically stable and is both controllable and observable, then the ARE has a unique positive definite solution $\mathbf{P}$, which minimizes the performance index J, when used in the optimum control law of equation 1.4. This approach is often used to generate a non-time varying control law that produces near optimal results.

The LQR design, while extremely robust from a theoritical standpoint, requires full state feedback which is often impractical in the implementation or actual systems. Optimal state estimation, the *dual problem* to the optimal control formulation, usually results in the implementation of a Kalman filter to provide estimates of missing or noisy states in the feedback path. This configuration is known as leading to the Linear Quadratic Gaussian (LQG) controller. In practice however, the LQG design results in a controller that is notoriously non-robust.

The Nelder-Meade algorithm, is a numerical search technique that does not require analytical differentiation [12]. It is effective and computationally compact, however, it is not guaranteed to find *global* optimums. Instead, it must be initialized with an initial guess, and tends to find the *local* minimum of a function, in the

vicinity of the initial guess. The Nelder-Meade algorithm is formulated in terms of a *simplex*, which is a generalized triangle of N dimensions. For functions of two variables, the simplex is just a triangle. The algorithm works by evaluating the function to be optimized at the three vertices of the triangle, ranking the vertices from best to worst. The worst vertex is rejected, and a method is provided for computing a new vertex so that a sequence of triangles is formed that will converge on the minimum of the function. The simplex concept easily extends the mechanization of the algorithm to include functions of N variables.

Modern control theory optimization techniques, are limited in the ways in which system performance and robustness characteristics must be specified. Most control system engineers will be faced with many design problems in which the given specifications cannot readily be formulated in terms required by the available theoretic design criteria. In particular, many long accepted norms of classical system performance criteria, such as maximum overshoot or settling time, may result in relationships which do not have analytic solutions. In many other situations, desirable performance or robustness criteria may result in relationships between controller parameters and the performance index which are either non-differentiable or non-convex. Either situation makes solution by standard numerical methods difficult. In particular, numerical search algorithms tend to get stuck on local maxima or minima, while enumerative or random search techniques take too long.

1.4 An Introduction to Genetic Algorithms

Genetic algorithms are a numerical search method which mathematically mimics some of the aspects of biological natural selection in order to achieve adaptation to a specific measure of fitness. Genetic algorithms, or GA's, first appeared in the early

1970's and were largly developed by John Holland, his colleagues and students at the University of Michigan [7]. Being non-calculas based, GA's do not require derivative information about the design specification. Instead they use a biological approach to randomly mating individual *genetically coded* characteristics, with preference weighted toward those which produce individuals displaying the greatest *fitness* . While employing random choice as a tool in the process, genetic algorithms are not purely random search techniques, but rather exploit the surprisingly powerful optimization tendency of population group evolution. Of particular importance is the fact that genetic algorithms work well in finding *global* optimums and are much less likely to converge on local maxima or minima in the manner of many numerical search methods.

Genetic algorithms may be constructed of varying degrees of complexity, but an effective form can be fairly easy to implement. First, a fitness function to be maximized by the algorithm must be determined for the problem at hand. In the case of an LQR control system, for example, this fitness function could be related to the quadratic performance index. In this case the fitness function must express an inverse relationship to the quadratic performance index, since the LQR solution works to minimize this index, while the genetic algorithm, for technical reasons that will become apparent, works to maximize the fitness function.

Once the fitness function is determined, the variable parameters, such as the control system feedback gains, are coded into bit strings consisting of ones and zeros. An initial population of random strings is generated, and the system fitness function is evaluated for each individual in the population. Successive generations are then generated by the operations of: *Reproduction* , *Crossover* and *Mutation* , which are explained in greater detail in Chapter 3.

The processes of reproduction, crossover and mutation are repeated until an

entirely new population has been generated. The old generation is discarded, and the fitness of each individual in the new generation is determined. The sum of the fitness values is calculated for the new generation, and reproduction, crossover and mutation are ready to begin again. Statistics are maintained to monitor such things as the minimum, maximum, and average fitness values for each generation. The process is halted when an appropriate *convergence* criteria, such as the average fitness value for a generation, reaches a specified level.

While the mechanics of genetics algorithms are fairly easy to understand, the underlying reasons for their success are more subtle. This phenomenon is explained by the concept of schemata templates and the building block hypothesis. Binary schemata templates are those in which some bit positions are important, while other bit positions are unimportant, or *don't care* bits. For example the schemata template 10*1**0, may be thought of as having 4 important bits, with 3 intervening *don't care* bits. For this schemata, the actual bit strings 1001000, 1011100, and 1001010 would all have an equivalent value, while the bit string 1010110 would have an inferior value.

It is postulated that useful schemata, which result in higher fitness values, concentrate in short building blocks. These building blocks are then sampled and recombined by the operations of reproduction and crossover at a much higher frequency than those of inferior combinations. The mutation operation assists by introducing new information at a rate that will not disrupt the building block process. Goldberg works through some examples to illustrate this process in reference [7], and in reference [8, page 1570] he asserts that:

> Holland's schema theorem places the theory of genetic algorithms on rigorous footing by calculating a bound on the growth of useful similarities or *building blocks* . Moreover, it has been recently conjectured,

that global convergence of certain types of GA is probabilistically polynomial. If shown correct, this conjecture could revolutionize the global solution of almost any problem.

1.5 Optimization via Genetic Algorithm - An Example

A simple optimization problem with one independent parameter will now be used to demonstrate the genetic algorithm explained in the previous section. Assuming that the value of the fitness function is represented exactly by the following equation in the parameter x :

$$fitness = 10.0 - (x - 2.5)^2 + 5\,sin(4\pi x) \tag{1.7}$$

which is plotted in figure 1.1.

While simple in construction, the closely spaced relative maxima and minima in this example present a non-trivial optimization problem for most search techniques. Using MATLAB to solve this function numerically for the three highest peaks yields the following:

x value	peak fitness
2.1245	14.8589
2.6243	14.9844
3.1242	14.6101

Thus for this example, the maximum fitness occurs at the center peak with a value of 14.9844, and the optimum value of x is 2.6243.

Using a population size of 20, and a 16 bit string to represent values of x ranging from 0.0 to 5.0, nine generations of the genetic algorithm were performed

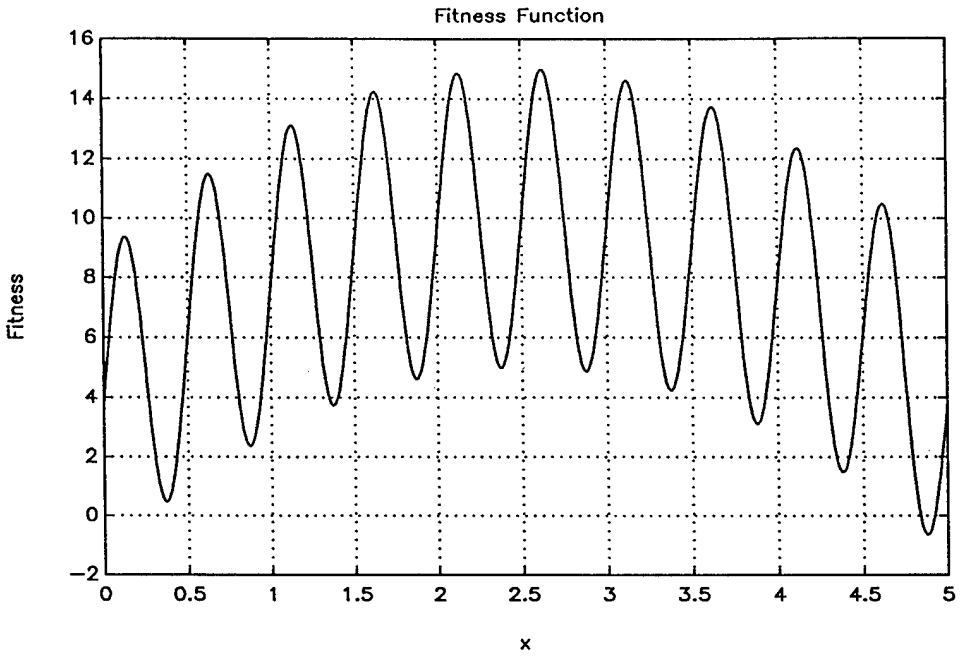

Figure 1.1: Fitness Function for Genetic Algorithm Example

using the MATLAB program GATEST, which is described in the Appendix. Figure 1.2 shows the initial population of random values for generation number 1. The four plots in this figure are generated for each generation and show the fitness and parameter values for each individual in the left hand plots, as well as maximum fitness and average fitness statistics for each generation in the right hand plots. For the initial generation, there is no statistical data available for the right hand plots.

Figure 1.3 shows the results after five generations of reproduction, crossover and mutation, which produces an optimum value for x of 2.6397, correct to within .58%. Figure 1.4 then shows the results after nine generations, which produces an optimum value for x of 2.6276, correct to within .12%. Figures 1.5 and 1.6 show in greater detail the history of the maximum fitness value and best, or *optimum* , parameter with increasing generation.

Note that five generations represent only 100 trial parameters, and nine

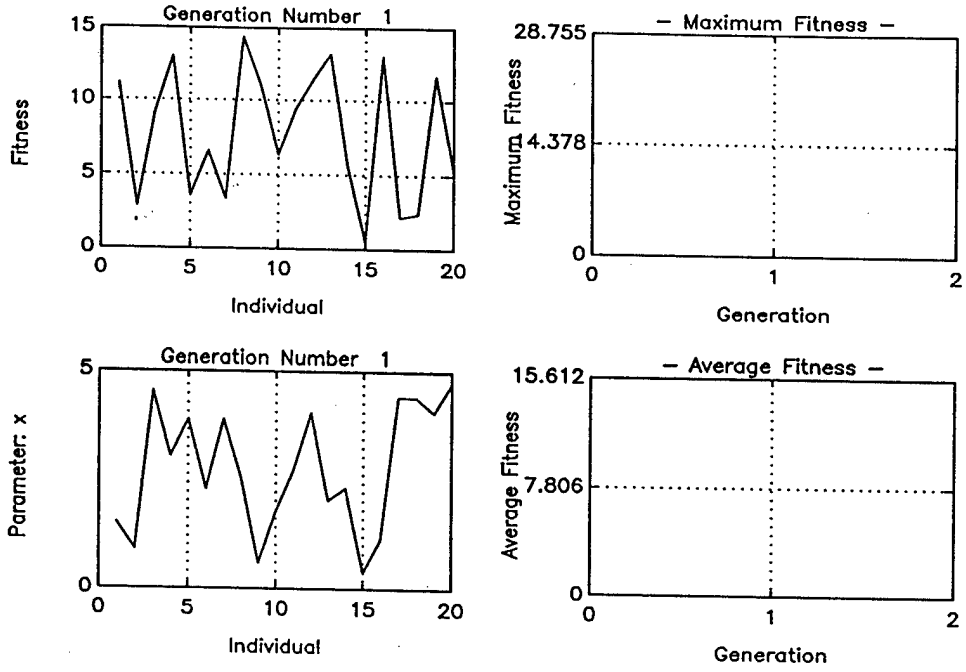

Figure 1.2: Initial Generation for Genetic Algorithm Example

generations represent only 180 trials. For comparison, a purely random search would require something on the order of 5000 trials to achieve an accuracy of .1% over the same range, and most gradient search techniques would have extreme difficulty optimizing this function.

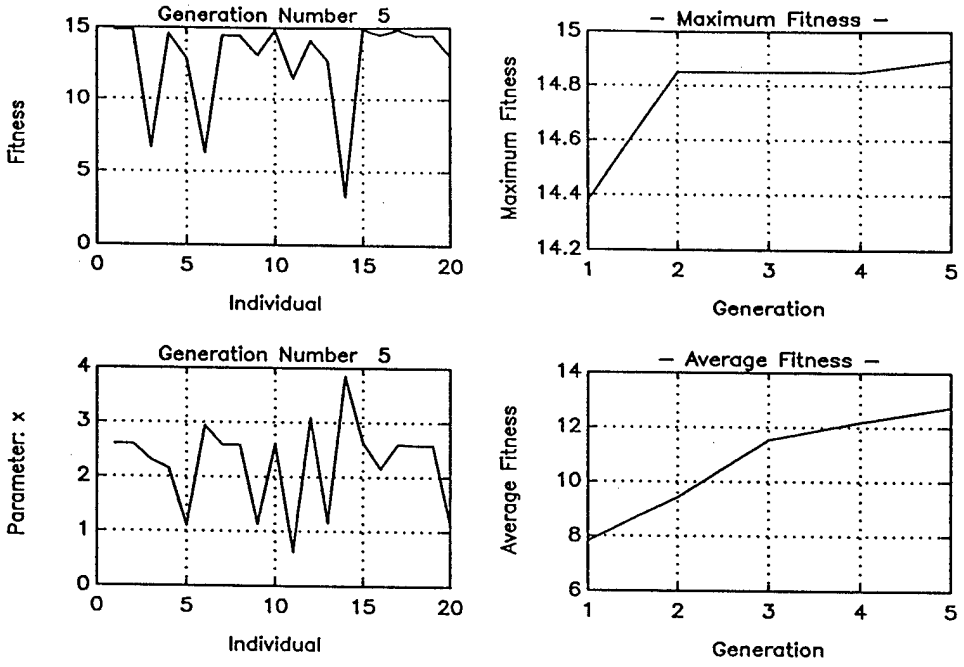

Figure 1.3: Results After Five Generations for Genetic Algorithm Example

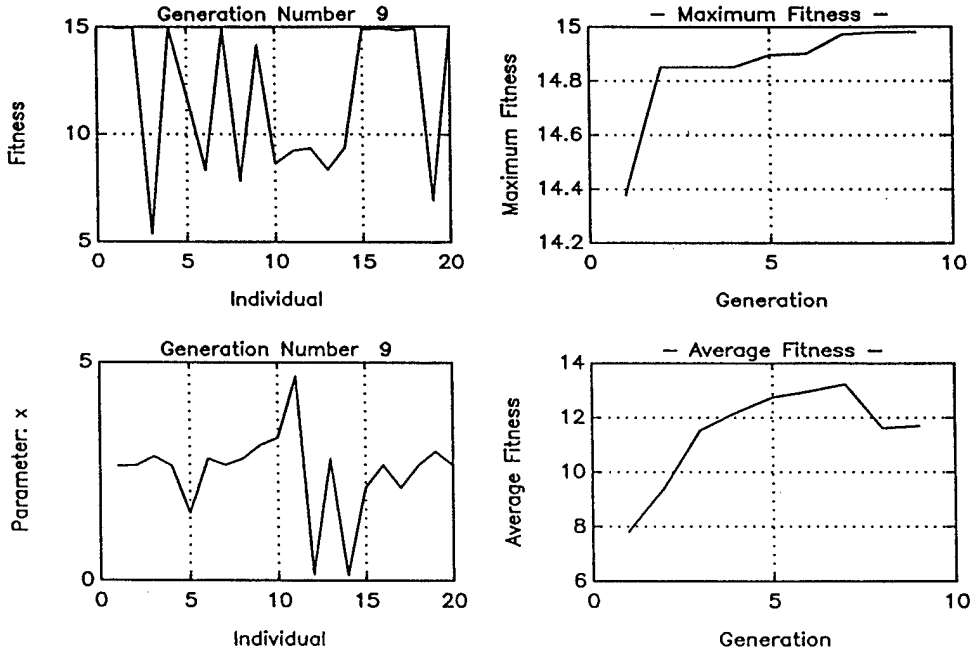

Figure 1.4: Results After Nine Generations for Genetic Algorithm Example

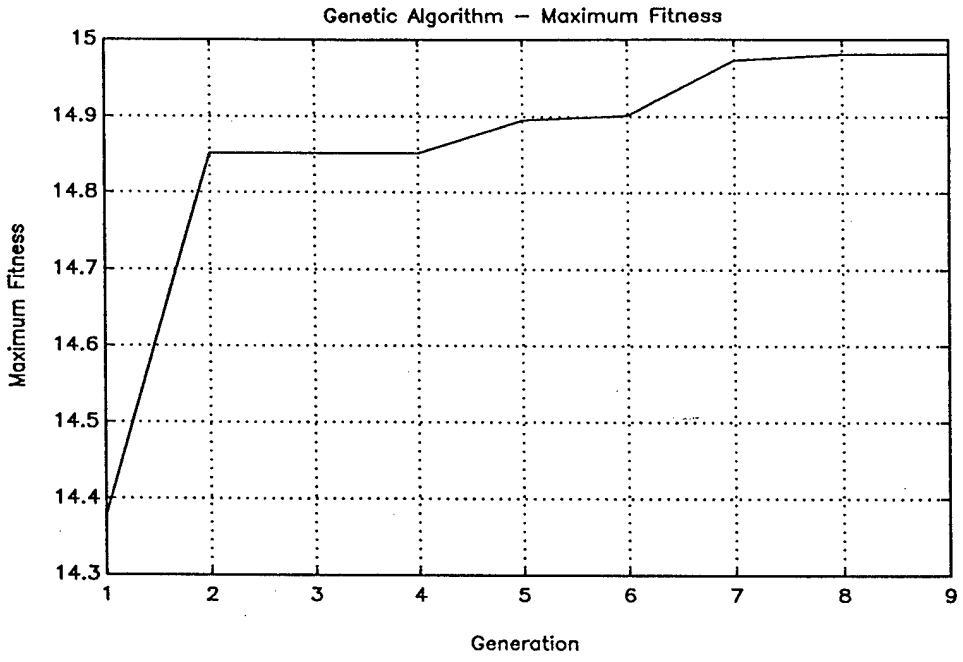

Figure 1.5: Maximum Fitness vs Generation for Genetic Algorithm Example

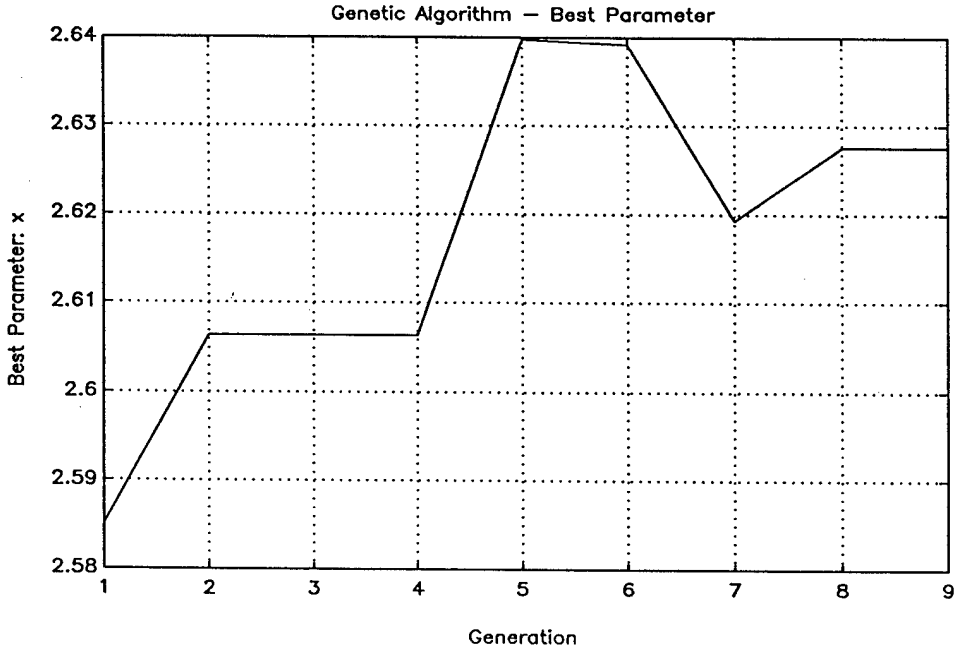

Figure 1.6: Best parameter vs generation for Genetic Algorithm example

CHAPTER 2

APPLICATION SYSTEM MODEL

A general feedback control scheme for a pitch plane missile autopilot controller is shown in figure 2.1. The sensors provide feedback variables to the controller which may be some combination of state and response variables. Regarding the pitch plane missile dynamics, most actual missile autopilot controllers accept normal acceleration commands (η_c) from the guidance loop, and utilize accelerometer (η_s) and rate gyro ($\dot{\theta}_s$) feedback information to determine the actuator command (δ_c).

The actual airframe and actuator and even sensor dynamics may be highly non-linear over the flight regime of any missile. However, over small ranges of flight conditions, where parameters such as Mach number, dynamic pressure, fuel weight, thrust, etc., may be considered constant, linearized models can be developed for

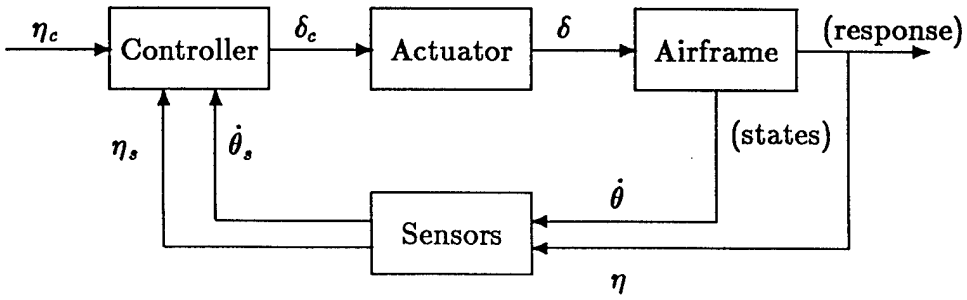

Figure 2.1: Missile Autopilot Feedback Control Scheme

use in the control system synthesis. These models can produce controllers that are effective within some range of the assumed flight conditions. Usually several conditions are studied, and the resulting controller parmeters are then *scheduled* over the desired flight envelope.

The approach used in this thesis will be to develop a linearized airframe and actuator model and assume hypothetical values that correspond to a single flight condition. A state space model of this system will be developed, with the states being:

$$\begin{aligned} x_1 &= \alpha &&= \text{angle of attack} \\ x_2 &= \dot{\theta} &&= \text{pitch angular rate} \\ x_3 &= \delta &&= \text{actuator deflection} \end{aligned}$$

Full state feedback will be assumed, with angle of attack (α) used in place of normal acceleration (η) as the autopilot tracking variable. This is done for simplification in the development of the system model, but in no way compromises the applicability of the results, since it will be seen that in the linear model η is just a linear combination of α and δ. For simplicity also, the sensor dynamics will be ignored, and the sensor transfer functions will assumed to be unity so that the feedback states are the actual modelled states.

The following sections develop the equations for the linearized airframe and actuator models and provide the hypothetical flight dependent parameters used in the ensuing analysis. A state space system model is then developed, along with the discrete time model required for digital simulation. Finally, the measures of performance that will be used in the optimization studies are developed as applied to this model.

2.1 Linearized Missile Airframe Dynamics

The 6 degree-of-freedom linearized aerodynamics model for a missile (or aircraft) rigid body are developed in section 2.6 of reference [5]. We will extract the pitch plane equations from this development, with some further simplifications and changes in nomenclature. First we define the pitch plane dynamic variables:

α - angle of attack
θ - pitch angle
γ - flight path angle
V - velocity vector
η - body normal acceleration
δ - actuator deflection angle
x - body centerline axis
z - body normal axis

as shown in figure 2.2.

Beginning with the three axes force and moment equations, reference [5] proceeds to the linearized equations for small perturbations about an operating condition by assuming constant velocity and attitude. Control surfaces and engine thrust are also assumed to be trimmed to these conditions. Longitudinal and lateral dynamics, which are normally only lightly coupled, are assumed uncoupled so that control system design can proceed independently in each channel. The resulting three equations for the longitudinal (pitch) axis are repeated below, but with a small change in nomenclature to use A for the axial force (instead of X), N for the normal force (instead of Z), $\dot{\theta}$ for the pitch angular rate (instead of q), and δ for the elevator (or actuator) deflection angle (instead of δ_E) [5, page 45]:

$$\delta\dot{u} = A_u \Delta u + A_\alpha \alpha - g\theta + A_E \delta \tag{2.1}$$

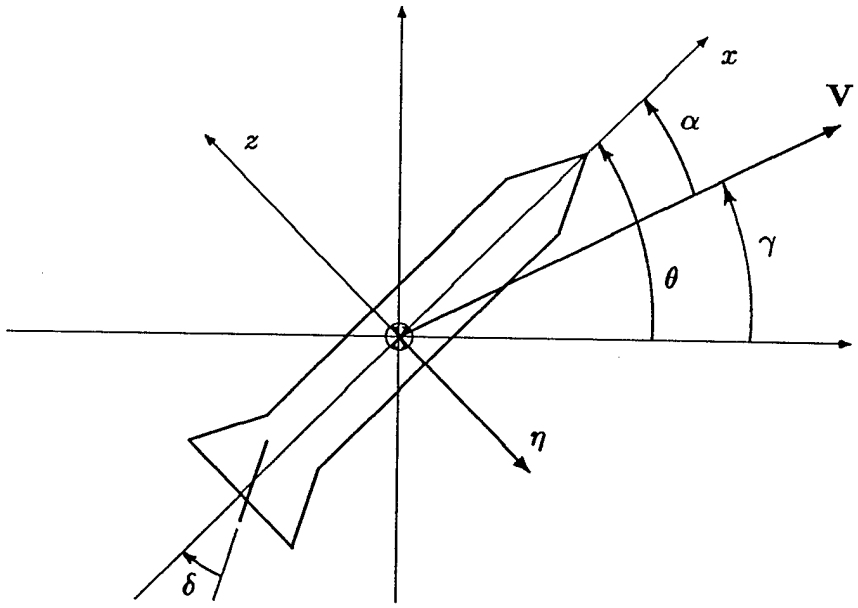

Figure 2.2: Missile Pitch Plane Dynamic Variables

$$\dot{\alpha} = (N_u/V)\Delta u + (N_\alpha/V)\alpha + \dot{\theta} + (N_E/V)\delta \tag{2.2}$$

$$\ddot{\theta} = M_u\Delta u + M_\alpha\alpha + M_q\dot{\theta} + M_E\delta \tag{2.3}$$

where the variables are defined as follows:

A	is the total force vector acting along the body x axis (axial force)
N	is the total force vector acting along the body z axis (normal force)
M	is the total body pitching moment
Δu	is the change in the velocity vector ($\dot{\mathbf{V}}$)
$\dot{\theta}$	is the pitch angular rate
δ	is the actuator deflection angle

Note that in these equations, partial derivatives of the force and moment terms are used. In aerodynamics, these forces and moments are usually expressed in terms of the *dimensionless* aerodynamic coefficients, C_N, C_A, C_m in the following

manner:

$$N = QS_{\pi}C_N \tag{2.4}$$

$$A = QS_{\pi}C_A \tag{2.5}$$

$$M = QS_{\pi}dC_m \tag{2.6}$$

where:

- Q is the dynamic pressure (force/area)
- S_{π} is a reference area that is airframe dependent
- d is a reference length that is airframe dependent

The dynamic pressure, Q, is a function of air density and missile velocity ($Q = 1/2\rho V^2$) , and the aerodynamic coefficients are airframe dependent, and may also be functions of parameters such as angle of attack, Mach number, actuator deflection, and pitch rate. Thus the small perturbation derivative approach greatly simplifies an otherwise highly non-linear problem. Defining the following partial derivatives of the aerodynamic coefficients [15, page 66]:

$$C_{N\alpha} = \partial C_N/\partial\alpha \tag{2.7}$$

$$C_{N\delta} = \partial C_N/\partial\delta \tag{2.8}$$

$$C_{m\alpha} = \partial C_m/\partial\alpha \tag{2.9}$$

$$C_{m\delta} = \partial C_m/\partial\delta \tag{2.10}$$

$$C_{m\dot{\theta}} = \partial C_m / \partial \dot{\theta} \tag{2.11}$$

Finally the force and moment derivatives used in the linearized longitudinal equations can be defined as:

$$N_\alpha = (QS_\pi / mV)(C_{N\alpha} - C_A) \tag{2.12}$$

$$N_\delta = (QS_\pi / mV) C_{N\delta} \tag{2.13}$$

$$M_\alpha = (QS_\pi d / I_y) C_{m\alpha} \tag{2.14}$$

$$M_\delta = (QS_\pi d / I_y) C_{m\delta} \tag{2.15}$$

$$M_{\dot{\theta}} = (QS_{pi} d^2) / (2I_y V) C_{m\dot{\theta}} \tag{2.16}$$

For this thesis, we are primarily concerned with the angular rate equations, and will make the additional simplifying assumptions:

1. velocity is constant ($\Delta u = 0$)
2. gravity can be ignored ($g = 0$)
3. aerodynamic damping is negligible ($C_{m\dot{\theta}} \approx 0$)

Applying these assumptions, and manipulating V, yields the following simplified linear angular rate equations:

$$\dot{\alpha} = N_\alpha \alpha + \dot{\theta} + N_\delta \delta \tag{2.17}$$

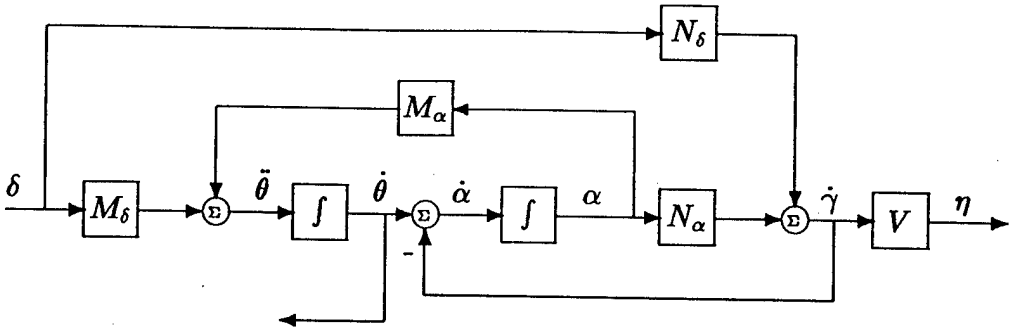

Figure 2.3: Linearized Airframe Model Block Diagram

$$\ddot{\theta} = \dot{q} = M_\alpha \alpha + M_{\dot{\theta}} \dot{\theta} + M_\delta \delta \tag{2.18}$$

As stated previously, angle of attack will be used as the control system tracking variable in this thesis. It should be noted that for applications where the normal acceleration response (η) is required, a good approximation is obtained by assuming:

$$\eta = V\dot{\gamma} = V(N_\alpha \alpha + N_\delta \delta) \tag{2.19}$$

Equations 2.17 , 2.18 and 2.19 will become the linearized airframe model for this thesis, and are represented in block diagram form in figure 2.3

Values for the linearized aerodynamic constant terms are assumed for purposes of analysis in this thesis that represent a fictitious but unstable airframe at some hypothetical flight condition as follows:

Assumed Flight Condition Constants

$M_\alpha = 64.11$ $\quad N_\alpha = .1803$

$M_\delta = -62.34$ $\quad N_\delta = .0738$

$M_{\dot{\theta}} = 0$ $\quad V = 892$ m/sec

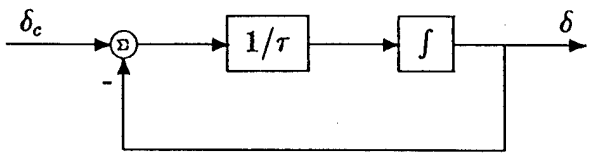

Figure 2.4: Linear Actuator Model Block Diagram.

2.2 Linear Actuator Model

A simple actuator with linear response characteristics can be modelled by the following differential equation:

$$\dot{\delta} = (1/\tau)(\delta_c - \delta) \tag{2.20}$$

where:

- δ_c is the commanded actuator angular deflection
- δ is the actuator response
- τ is the actuator time constant

This linear actuator model is also represented in block diagram form as shown in figure 2.4

For purposes of analysis in this thesis, the following actuator time constant will be used:

$$\tau = .02 seconds \tag{2.21}$$

2.3 State Space System Model

Combining the linearized airframe and linear actuator models, the state space system model is developed by making the following state variable assignments:

$x_1 = \alpha$ = angle of attack

$x_2 = \dot{\theta}$ = pitch angular rate

$x_3 = \delta$ = actuator angular deflection

Using the results of the previous two sections, and making the appropriate state variable substitutions, the combined state equations are:

$$\dot{x_1} = N_\alpha x_1 + x_2 + N_\delta x_3 \tag{2.22}$$

$$\dot{x_2} = M_\alpha x_1 + M_\delta x_3 \tag{2.23}$$

$$\dot{x_3} = -(1/\tau)x_3 + (1/\tau)\delta_c \tag{2.24}$$

and the output equation for η is:

$$y = V N_\alpha x_1 + V N_\delta \delta \tag{2.25}$$

This set of linear differential equations can be written in the familiar matrix notation as:

$$\dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{Bu} \tag{2.26}$$

$$\mathbf{y} = \mathbf{Cx} + \mathbf{Du} \tag{2.27}$$

where:

- **x** is the 3 x 1 state vector
- **u** is the 1 x 1 control variable (a scaler in this case)
- **A** is the 3 x 3 system matrix
- **B** is the 3 x 1 control matrix
- **C** is the 1 x 3 state output connection matrix
- **D** is the 1 x 1 control output connection matrix

In terms of the previously defined variables, these matrices for the state space system model are:

$$\mathbf{A} = \begin{bmatrix} N_\alpha & 1.0 & N_\delta \\ M_\alpha & 0.0 & M_\delta \\ 0.0 & 0.0 & -(1/\tau) \end{bmatrix} \tag{2.28}$$

$$\mathbf{B} = \begin{bmatrix} 0.0 \\ 0.0 \\ 1/\tau \end{bmatrix} \tag{2.29}$$

$$\mathbf{C} = \begin{bmatrix} VN_\alpha & 0.0 & VN_\delta \end{bmatrix} \tag{2.30}$$

$$\mathbf{D} = [0.0] \tag{2.31}$$

2.4 Linear Feedback Controller

The linear feedback controller provides one or more control variables that are formed as a linear combination of the system states, as expressed in the following equation:

$$u = -(k_1 x_1 + k_2 x_2 + k_3 x_3) \tag{2.32}$$

or written in matrix form as:

$$\mathbf{u} = -\mathbf{k}\,\mathbf{x} \tag{2.33}$$

The gain vector $\mathbf{k}$ consists of the three feedback gains that are to be determined by the control system design or optimization process.

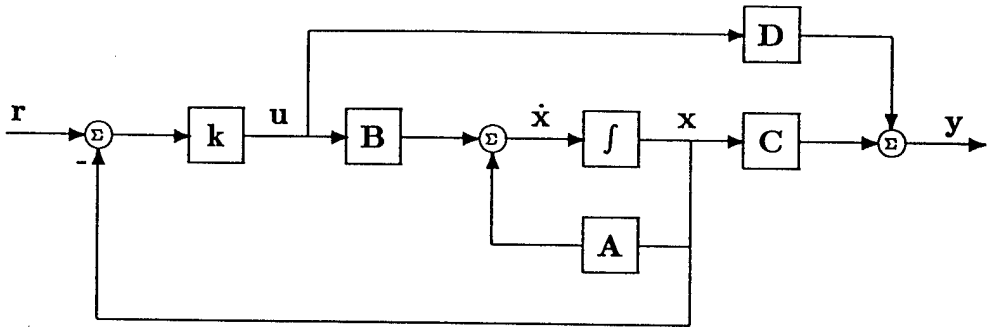

Figure 2.5: Closed Loop Linear Control System

This notation is extended to tracking systems by including the non-zero input command vector, **r**, as follows:

$$\mathbf{u} = \mathbf{k}\,(\mathbf{r} - \mathbf{x}) \tag{2.34}$$

where:

- **r** is the 3 x 1 input state command vector
- **x** is the 3 x 1 state vector
- **u** is the 1 x 1 control variable (a scaler in this case)
- **k** is the 1 x 3 gain vector

The resulting closed loop system, when the linear feedback controller is applied to the linear system model, is shown in the block diagram of figure 2.5

For the closed loop linear control system of figure 2.5 the state and output equations can be expressed as:

$$\dot{\mathbf{x}} = \mathbf{A_c x} + \mathbf{B_c r} \tag{2.35}$$

$$\mathbf{y} = \mathbf{C_c x} + \mathbf{D_c r} \tag{2.36}$$

where the closed loop system matrices are defined in terms of the open loop system

matrices and feedback gains by the following equations:

$$\mathbf{A_c} = \mathbf{A} - \mathbf{Bk} \tag{2.37}$$

$$\mathbf{B_c} = \mathbf{Bk} \tag{2.38}$$

$$\mathbf{C_c} = \mathbf{C} \tag{2.39}$$

$$\mathbf{D_c} = \mathbf{D} \tag{2.40}$$

2.5 Discrete Time Simulation Model

Equations 2.35 through 2.40 establish the closed loop system equations in *continuous* form. For purposes of digital simulation and evaluation, however, we wish to determine the corresponding system equations in *discrete* form. That is, given the closed loop continuous system matrices, and a simulation time interval ($d\tau$), we wish to determine the corresponding discrete matrices for the following equations:

$$\mathbf{x}(k+1) = \mathbf{A_d}\mathbf{x}(k) + \mathbf{B_d}\mathbf{r}(k) \tag{2.41}$$

$$\mathbf{y}(k) = \mathbf{C_d}\mathbf{x}(k) + \mathbf{D_d}\mathbf{r}(k) \tag{2.42}$$

where $\mathbf{x}(k)$, $\mathbf{r}(k)$ and $\mathbf{y}(k)$ represent the state, input, and output variables at the k^{th} iteration, and $\mathbf{x}(k+1)$ represents the state variables at the $(k+1)^{th}$ iteration. There are many ways to convert from the continuous to the discrete system equation, but a standard method recommended in several of the references [14,6,10], and used in

this thesis involves the *state transition matrix* Φ. A complete derivation is given in [14, section 12.9], with the following results:

$$\begin{aligned} \mathbf{A_d} &= \Phi(d\tau) \\ &= \epsilon^{\mathbf{A_c} d\tau} \\ &= \mathbf{I} + \mathbf{A_c} d\tau + \mathbf{A_c}^2 d\tau^2/2! + ... \end{aligned} \tag{2.43}$$

$$\begin{aligned} \mathbf{B_d} &= \left[\int_0^{d\tau} \Phi(d\tau - t)dt\right] \mathbf{B_c} \\ &= \left[\mathbf{I}d\tau + \mathbf{A_c}d\tau^2/2! + \mathbf{A_c}^2 d\tau^3/3! + ...\right] \mathbf{B_c} \end{aligned} \tag{2.44}$$

$$\mathbf{C_d} = \mathbf{C_c} \tag{2.45}$$

$$\mathbf{D_d} = \mathbf{D_c} \tag{2.46}$$

The equations 2.41 through 2.46 represent the discrete time simulation model formulation for tracking systems as used in this thesis. Initial values for all states are assumed to be zero.

2.6 Measures of Performance

The specific measures of performance that will be used in forming the objective of the optimization problem in this thesis are:

1. quadratic performance index

2. rise time specification

3. peak actuator effort specification

4. settling error specification

Each of these measures is generated from the time response of the discrete time simulation model to a unit step input in angle of attack. These measures of performance are used, either individually or in weighted combinations, to create a single cost function that can be related to a fitness function for use in the genetic algorithm.

The quadratic performance index used is the same as that of the Linear Quadratic Regulator problem, which was given previously in continuous form in equaton 1.1. For the discrete time simulation model, this performance index is computed from the system discrete time response by the following equations:

$$\mathbf{j}(k) = [\mathbf{r}(k) - \mathbf{x}(k)]' \, \mathbf{Q} \, [\mathbf{r}(k) - \mathbf{x}(k)] + \mathbf{u}'(k)\mathbf{R}\mathbf{u}(k) \tag{2.47}$$

$$J = \sum_{k=0}^{N} .5 \, d\tau \, [\mathbf{j}(k) + \mathbf{j}(k-1)] \tag{2.48}$$

where:

k	is the discrete iteration counter (time)
x(k)	is the state variable vector,
r(k)	is the input (tracking) variable vector,
u(k)	is the control variable vector,
Q	is a positive semi-definite weighting matrix
R	is a positive definite weighting matrix
N	is the number of discrete iterations (corresponding to the final time)

Values used for the state and controls weighting matrices are:

$$\mathbf{Q} = \begin{bmatrix} 10.0 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.01 \end{bmatrix} \tag{2.49}$$

$$\mathbf{R} = [0.01] \tag{2.50}$$

For the unit step input, the tracking variable is just the constant command:

$$\mathbf{r}'(k) = \begin{bmatrix} 1.0 & 0 & 0 \end{bmatrix} \tag{2.51}$$

for k = 1 ... N.

Rise time specifications vary in different applications and are often defined differently in various texts. The most widely accepted definition of rise time is the time it takes for a signal to go from 10% to 90% of its final value [14, page 124]. This definition is somewhat cumbersome to compute, however, so for purposes of this thesis the rise time, indicated by t_{rise}, will represent the time for the system angle of attack response (x_1) to reach 80% of its *commanded* final value.

The peak actuator effort specification is simply a measure of the peak value of the actuator response (x_3) to the unit step input in angle of attack, as indicated by the following expression:

$$\delta_{max} = max\{\,|x_3|\,\} \tag{2.52}$$

The settling error specification is a measure of the absolute value of the difference between the angle of attack and its commanded value at the finite final time ($T = N d\tau$). That is:

$$\epsilon_{settle} = |r_1(N) - x_1(N)| \tag{2.53}$$

CHAPTER 3

GENETIC ALGORITHM IMPLEMENTATION

The basic form of the genetic algorithm used in this thesis has been described previously in section 1.4. This chapter will now describe the specific implementation details of the genetic algorithm as applied to the control system optimimization problem of this thesis.

3.1 Fitness Function

The discrete quadratic performance index (J) given previously in equation 2.48 forms the basis for constructing a fitness function. As seen previously, the genetic algorithm acts to maximize the fitness function, while in fact we wish to minimize the quadratic performance index (J). We can accommodate these conflicting requirements by constructing a fitness function of the form:

$$fitness = \epsilon^{2/C} - 1 \tag{3.1}$$

where C is a cost term which is limited by:

$$0.5 \leq C \leq 100 \tag{3.2}$$

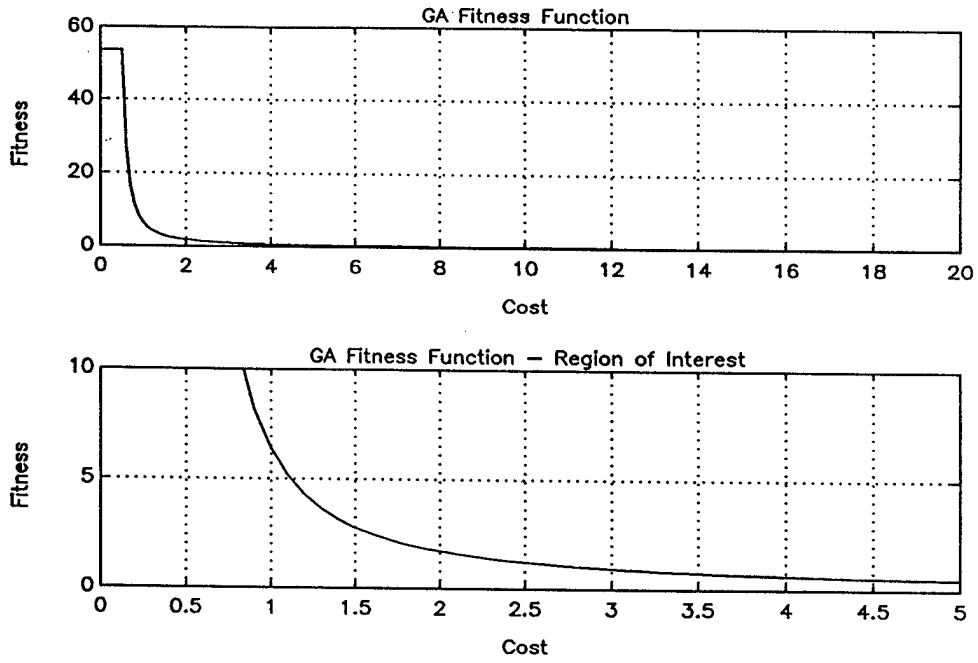

Figure 3.1: Genetic Algorithm Fitness Function vs Cost

The resulting function is plotted in figure 3.1.

The factor of 2 in the numerator of the exponential is somewhat arbitrary and can be adjusted to give the fitness function a good gradient in the region of interest for a specific problem. The value of 2 was determined after a few trials and was found to work well for the optimization problem of this thesis. The constant value 1 is subtracted from the exponential to shift the function to the origin for large cost terms ($C \Rightarrow \infty$), but does not really affect the optimization process.

For a simple quadratic performance specification, with no other constraints, the genetic algorithm fitness function is formed by setting the cost term in equation 3.1 equal to the quadratic performance index:

$$C = J \tag{3.3}$$

3.2 Performance Penalties

Additional design specification constraints can be incorporated into the genetic algorithm optimization problem by adding appropriately weighted performance penalties to the cost term C as shown in the following equation:

$$C = J + P_\delta + P_{settle} + P_{rise} \tag{3.4}$$

where:

P_δ	is an actuator peak response penalty
P_{settle}	is a settling error penalty
P_{rise}	is a rise time penalty

These performance penalties and their associated weighting factors are given by the following expressions:

$$P_{rise} = \begin{cases} 10.0\,(t_{rise} - \phi_{rise}) & \text{if } t_{rise} > \phi_{rise} \\ 0.0 & \text{otherwise} \end{cases} \tag{3.5}$$

$$P_\delta = \begin{cases} 20.0\,(\delta_{max} - \phi_\delta) & \text{if } \delta_{max} > \phi_\delta \\ 0.0 & \text{otherwise} \end{cases} \tag{3.6}$$

$$P_{settle} = \begin{cases} 10.0\,(\epsilon_{settle} - \phi_{settle}) & \text{if } \epsilon_{settle} > \phi_{settle} \\ 0.0 & \text{otherwise} \end{cases} \tag{3.7}$$

where:

ϕ_δ	is the design actuator peak response specification
ϕ_{rise}	is the design rise time specification
ϕ_{settle}	is the allowable settling error specification

3.3 Reproduction

Reproduction is the process of selecting two individuals from the current population for mating. Individuals are selected using a random function that gives preferential weighting to the individual's relative fitness within the population. The algorithm used in this thesis is similar to the simple genetic algorithm described in [7] and is implemented in the following manner:

Reproduction Algorithm

1. Determine the sum of the fitness values for the entire population.
2. Rank the individuals from highest to lowest fitness.
3. Determine the cumulative fitness values of each individual, ranked from highest to lowest. These numbers should range from zero to the sum of the fitness determined in step one.
4. Generate a random number from a uniform distribution ranging from zero to the sum of the fitness.
5. Select **parent 1** for mating whose cumulative fitness value is the nearest one less than or equal to the random number.
6. Remove **parent 1** from the available population for mating, and repeat the steps above to select **parent 2** .

3.4 Crossover

Crossover is the process of combining the parameter strings from two parents to form new *offspring* individuals. It is performed in a manner independent of the

actual numbers that those strings represent. For the multiparameter optimization problem of this thesis, the 48 bit parameter strings consist of three 16 bit words, each of which represents one of the gain parameters k_1, k_2 or k_3. Each parameter word is scaled separately, so that it can represent a value between the k_{lower} and k_{upper}, to the maximum resolution allowed by 16 bits. That is, the value of the least significant bit for each parameter word is given by:

$$k_{LSB} = (k_{upper} - k_{lower})/(2^{16} - 1) \tag{3.8}$$

In the genetic algorithm implemented in this thesis, two offspring (children) are generated from two parents in the following manner:

Crossover Algorithm

1. Generate a random number uniformly distributed between 0.0 and 1.0. If this number is less than the predetermined probability of crossover, p_{cross}, then copy the parent parameter word directly into the child, and skip to the next parameter word. Otherwise, continue.

2. Generate a random number uniformly distributed between 1 and the number of bits in a parameter word.

3. Swap the portion of the parameter words between the two parents that have bit locations higher than this random number.

4. Repeat the above steps for every parameter word in the parameter string.

The following example is given to illustrate this process for two single parameter word parent strings that are represented by the following bit sequences:

$$p_1 = 1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0 \tag{3.9}$$

$$p_2 = 1\,1\,1\,1\,0\,0\,0\,0\,1\,1\,1\,1\,0\,0\,0\,0 \tag{3.10}$$

Assuming that the random crossover site number is 6, the following two single parameter word child strings would be generated:

$$c_1 = 1\,0\,1\,0\,1\,0\,0\,0\,1\,1\,1\,1\,0\,0\,0\,0 \tag{3.11}$$

$$c_2 = 1\,1\,1\,1\,0\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0 \tag{3.12}$$

3.5 Mutation

Mutation is a further randomizing process that occurs at relatively low probability. After reproduction and crossover have generated an offspring, mutation may change the value of any bit in the resulting child strings from 0 to 1, or 1 to 0. Mutation can be implemented in several ways, but the straightforward method implemented in this thesis is as follows:

Mutation Algorithm

1. For each bit in the child parameter strings, generate a random number uniformly distributed beteen 0.0 and 1.0.

2. If the random number for any bit in the child parameter string is less than the predetermined probability of mutation, $p_{mutation}$, change the value of that bit. Otherwise leave the value of the bit unchanged.

3.6 Algorithm Enhancements

The genetic algorithm described in the previous sections follows closely the simple genetic algorithm described in [7]. There are, however, several notable differences, or enhancements, to the genetic algorithm in this thesis, which for this problem at least, improved its overall performance. These enhancements include:

Algorithm Improvements

1. Ensuring that no individual is mated with itself, if crossover is to be performed.

2. Copying the *best* 2 individuals directly into the next generation, without performing crossover or mutation. These individuals are still available for mating, but reduce the total number of matings required to fill the next generation. This ensures that the maximum fitness function is monotonic, and that the *best genes* are never lost.

3. Performing crossover separately on each parameter word in the multiparameter string.

4. Scaling each parameter word separately to take advantage of *a priori* information, increase accuracy, and accelerate convergence.

5. Modifying the random distribution of the crossover site selection to favor high order bits in early generations, gradually shifting to favor low order bits in later generations. The idea is to force the genetic algorithm to make big decisions early, then narrow its focus to fine tune the final answer. (This experimental scheme was implemented and showed promise, but was not used in the final optimization runs presented later in this thesis.)

It should be noted that several other algorithm *enhancements* were attempted during the course of this investigation that did not demonstrate any improvement in the performance of the algorithm. Some of these even degraded its overall performance. For completeness, the algorithm modifications which did not work are listed below:

Unsuccessful Algorithm Enhancements

1. Modifying the parameter scale factors as a function of generation number, attempting to *squeeze* the upper and lower limits toward the optimum values.

2. Saving the best individuals out of the combined old and new generations after each round of reproduction, crossover and mutation.

3. Generating 2 new individuals at random during each generation, and reducing the number of individuals generated by mating by 2.

4. Initializing a portion of the initial population to set strings in order to represent primitive Walsh function patterns in the initial population.

3.7 Convergence Criteria

In many optimization problems, it would be desired to determine a criteria for algorithm convergence. One such method, suggested in reference [8], is to terminate the algorithm when

1. The average fitness of the population is within 1% of the best fitness for that generation, and ...

2. a minimum of 20 generations have been examined.

In the example given in this reference for an LQR controller optimization problem, it is observed that this convergence criteria was reached after about 65 generations. It is also observed, however, that after about 15 generations, the best fitness is within a few percent of its final value.

In the controller optimization problem of this thesis, we are interested in obtaining the best fitness value and are less concerned with the average fitness evolution of the population taken as a whole. For this reason, as well as to conserve the computer time required in order to produce a given set of results, the genetic algorithm was executed for a fixed number of generations for each optimization run. It was found that 17 generations were sufficient to produce results within a few percent of the peak fitness values for the cases studied in this thesis.

CHAPTER 4

AUTOPILOT CONTROLLER OPTIMIZATION

This chapter presents the results of the genetic algorithm application to the missile autopilot control system optimization problem posed by this thesis. First, the standard steady state LQR solution is presented and the resulting closed loop system response analyzed. Results for the genetic algorithm optimization of the analagous problem, using a linear quadratic performance measure, are then presented and compared. Additional design specifications are next imposed to meet peak actuator effort, settling error, and rise time constraints. The results of the genetic algorithm optimization for each successive combination of constraints is presented, culminating in a controller which satisfies all of the constraints. It is posited that the resulting *optimal controller* is one for which analytic solutions are not available. Finally, the performance of the genetic algorithm is examined by comparing its solutions to a set of *true optimum* computer solutions found with the help of the Nelder-Meade numerical search algorithm. The results of the genetic algorithm optimization studies were used to initialize the Nelder-Meade algorithm, and a numerical search was continued to find the *true* optimum values.

The computer generated results presented in this thesis were obtained using

the MATLAB software package, operating on a 20 Mega Hertz 386 personal computer. MATLAB standard functions, as well as functions provided in the Control Toolbox and Robust Control Toolbox, were utilized. Several MATLAB programs and numerous special supporting functions were written in order to generate these results, and are provided in the Appendices for reference. The MATLAB function, FMINS, was used to perform the Nelder-Meade numerical optimization studies.

4.1 Steady State LQR Solution

The standard steady state Linear Quadratic Regulator solution to the continuous state space system model, described in section 2.3, was determined by solving the associated Algebraic Riccati equation. This was performed in MATLAB, using the functin LQR2 provided with the MATLAB Control Toolbox. The resulting feedback controller gains are shown below.

LQR Controller Gains

$$k_1 = -34.1723$$

$$k_2 = -3.6403$$

$$k_3 = 2.3434$$

The resulting closed loop discrete time system, which incorportes the LQR feedback gains, was constructed and analyzed using the MATLAB program AP5. Analysis of this analytic controller solution provides a confidence check for the simulation programs and serves as a baseline design for estimating appropriate ranges of the gain parameters to be used in the genetic algorithm.

The state variable time response of this system was generated by inserting a unit step input into the angle of attack command variable ($r_1 = 1$). The resulting *transient* response tends to be very active for the first few tenths of a second,

then settles toward the *steady state* values within a few seconds. For simulation purposes, a dual simulation time interval approach was implemented. First, the transient portion of step response was generated using a simulation time interval ($d\tau$) of .001 seconds, then the state transition matrix was recomputed, and the steady state portion of the step response was generated using a time interval of .01 seconds. This allows accurate integration during the period of high system dynamics, yet conserves on the total computer time required to evaluate a given configuration.

Figure 4.1 shows the actuator time response during the transient period from 0 to .2 seconds, from which the peak actuator response can be determined. Figure 4.2 shows the angle of attack response out to its assumed steady state value at 5.0 seconds. From this angle of attack history, the rise time and settling error parameters can be determined.

Bode plots for the open loop system, evaluated with the LQR feedback gains, are also obtained in AP5 by using the MATLAB Control Toolbox function BODE. Likewise, gain margin (G_{margin}), and phase margin (ϕ_{margin}) for this system are computed using the Toolbox function MARGIN. Figure 4.3 shows the Bode plots for the LQR feedback gain open loop system.

According to optimal control theory, the LQR controller design should exhibit an infinite gain margin and a minimum of 60 degrees of phase margin [1]. The phase margin can be obtained from the Bode plots, as the phase angle when the gain crosses 0 dB. Likewise, the gain margin is obtained from the Bode plots, as the gain value when the phase angle crosses 180 degrees. From these plots, it appears that the theoretical stability margins have been achieved. The phase margin is somewhat greater than 60 degrees, and the gain margin appears to be infinite, since the phase appears to approach 90 degrees asymptotically and will never cross 180

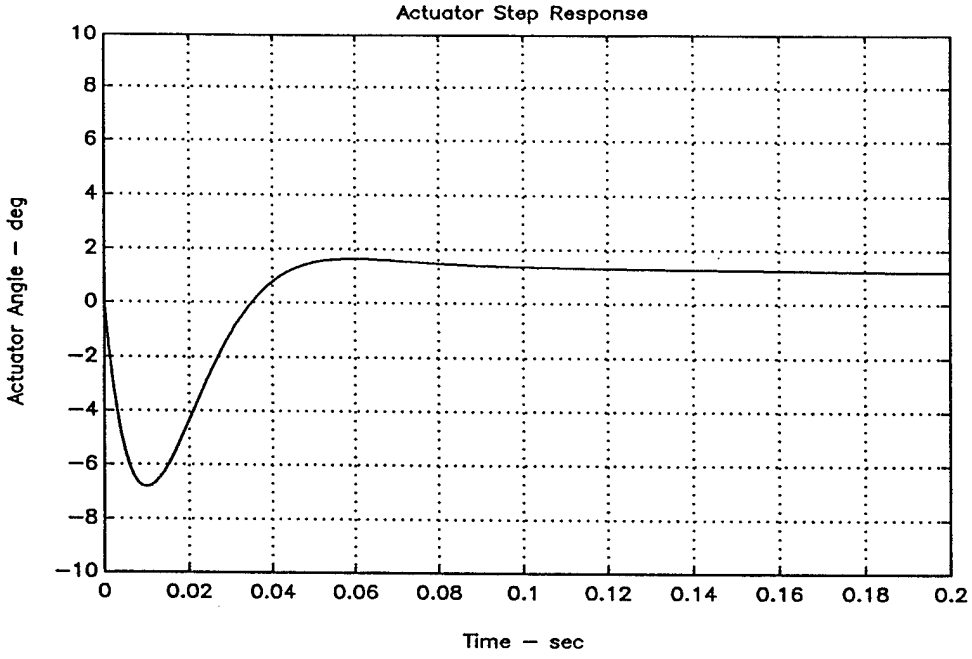

Figure 4.1: Actuator Response for the LQR Controller

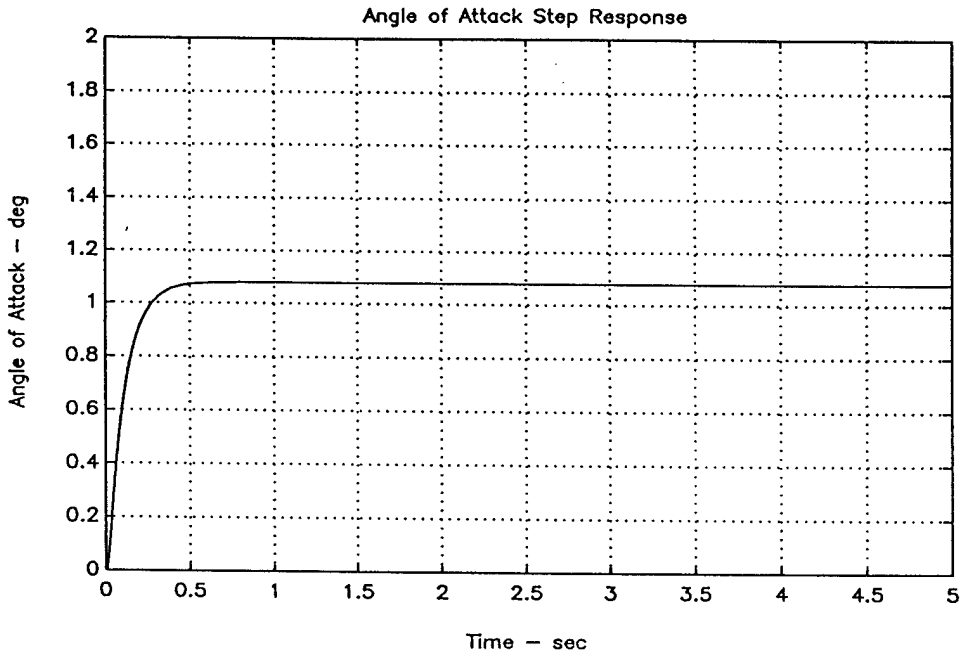

Figure 4.2: Angle of Attack Response for the LQR Controller

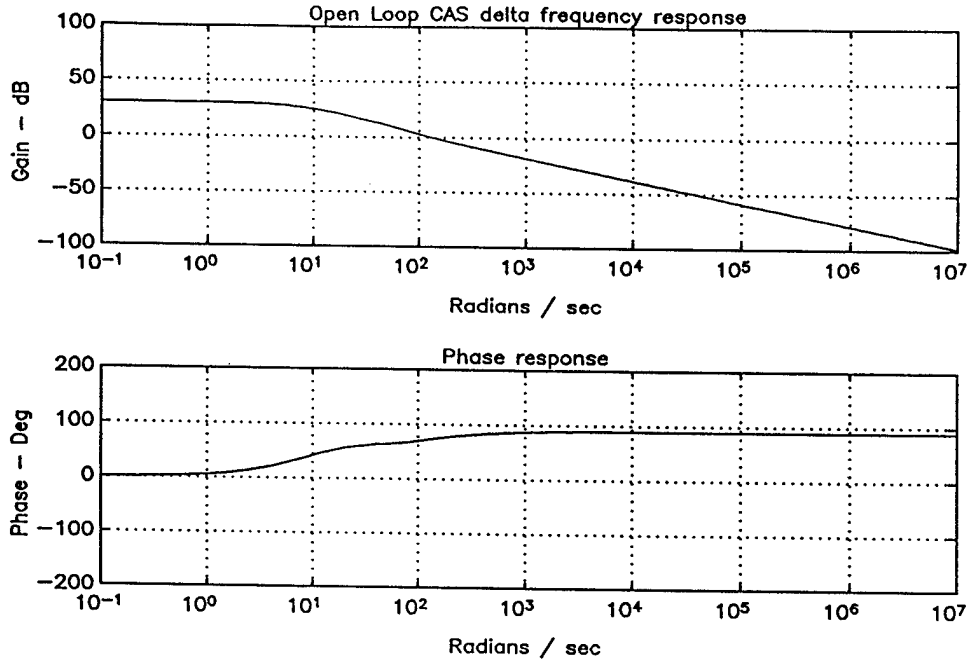

Figure 4.3: Bode Plots for the LQR Controller

degrees. Since no physical system can have infinite gain margin, we will henceforth indicate only that this gain margin exceeds 90 dB.

Using the computer to numerically analyze these time and frequency response histories, the following performance and stability margin characteristics are extracted and summarized for the LQR controller design.

LQR Controller Characteristics

δ_{max}	=	6.826
ϵ_{settle}	=	.079
t_{rise}	=	.151
ϕ_{margin}	=	72.77 deg.
G_{margin}	=	(> 90 dB)

4.2 Genetic Algorithm Optimization for the LQ Performance Index

The genetic algorithm described in Chapter 3 was implemented using the MATLAB program APGEN3, and configured to optimize for the Linear Quadratic performance index (J), described in section 2.6. This was accomplished by setting the GA cost function equal to the LQ performance index and computing the GA fitness function as described in section 3.1. The system resulting from this optimization problem will hence be referred to as the: *J* **Controller** . This is the genetic algorithm equivalent of the LQR design of the previous section, using identical **Q** and **R** weighting matrices. In this case, however, numerical analysis of the discrete time simulation response will be used to compute and optimize the GA fitness function. Differences are to be expected due to the discrete time simulation intervals (.001 and .01 seconds) and the finite integration period (5 seconds) used for evaluation of the performance index.

Snapshots of the genetic algorithm optimization process are shown in figures 4.4 through 4.8. The four plots in each of these figures are produced after each generation of 30 individuals, and record the following information:

1. The fitness value for each individual in the current generation.

2. The three gain parameter values (k_1, k_2, k_3) for each individual in the current generation.

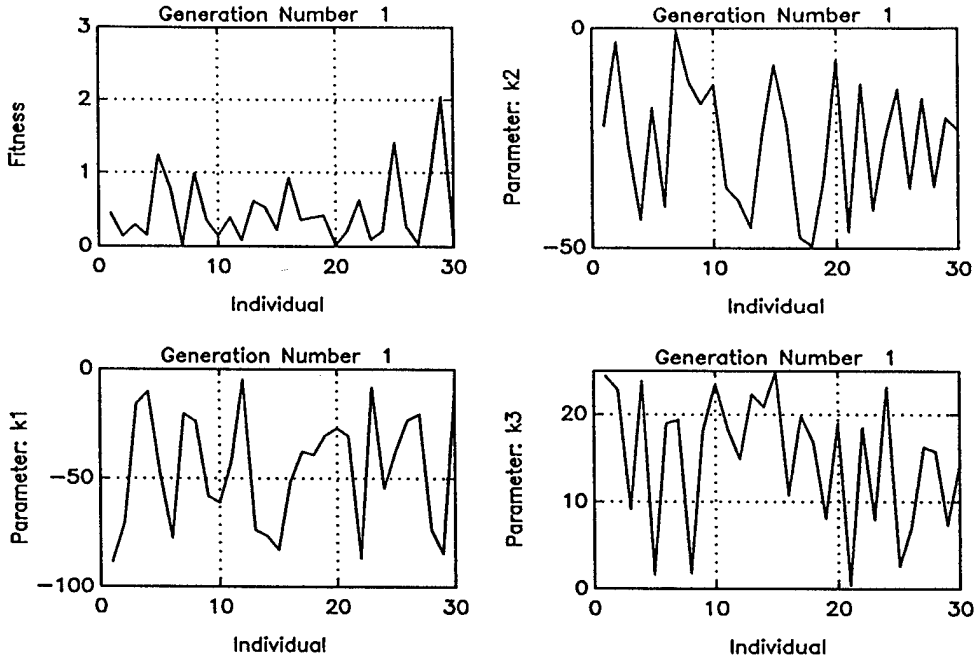

Figure 4.4: Genetic Algorithm Optimization - Initial Generation - J Controller

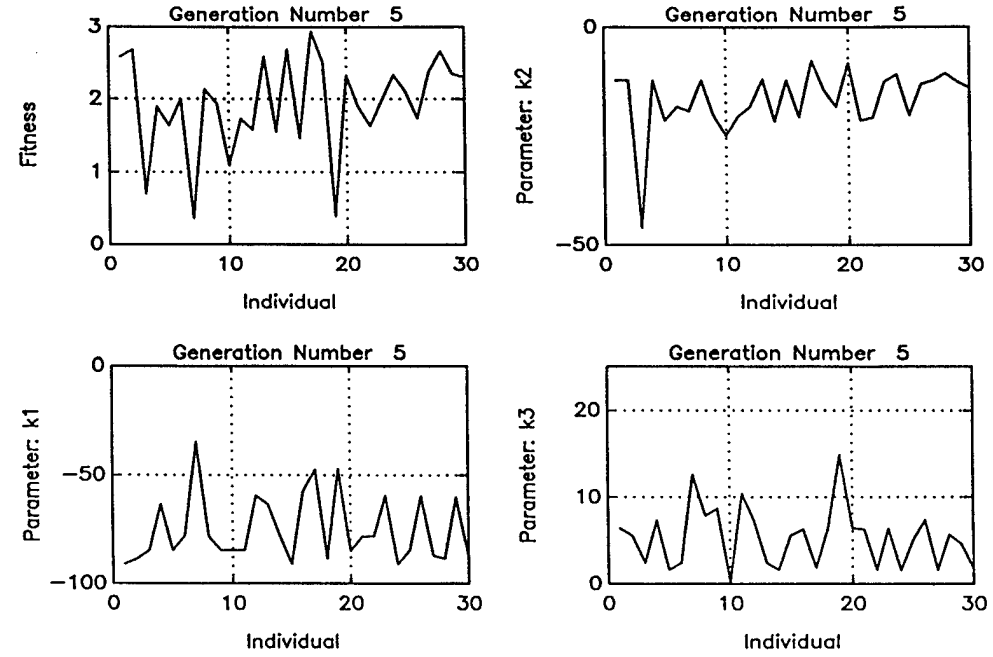

Figure 4.5: Genetic Algorithm Optimization - Generation 5 - J Controller

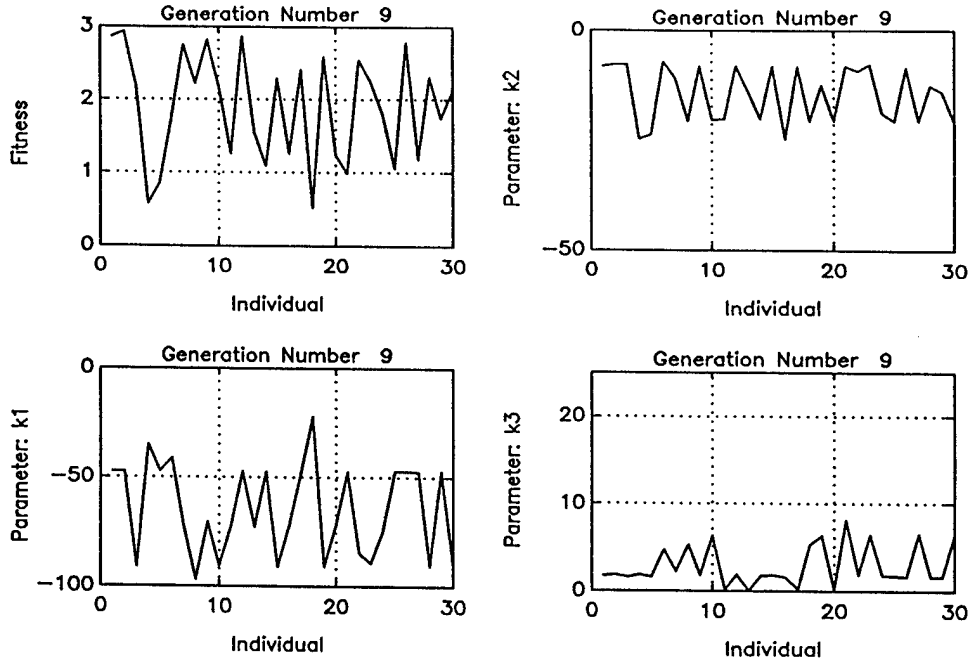

Figure 4.6: Genetic Algorithm Optimization - Generation 9 - J Controller

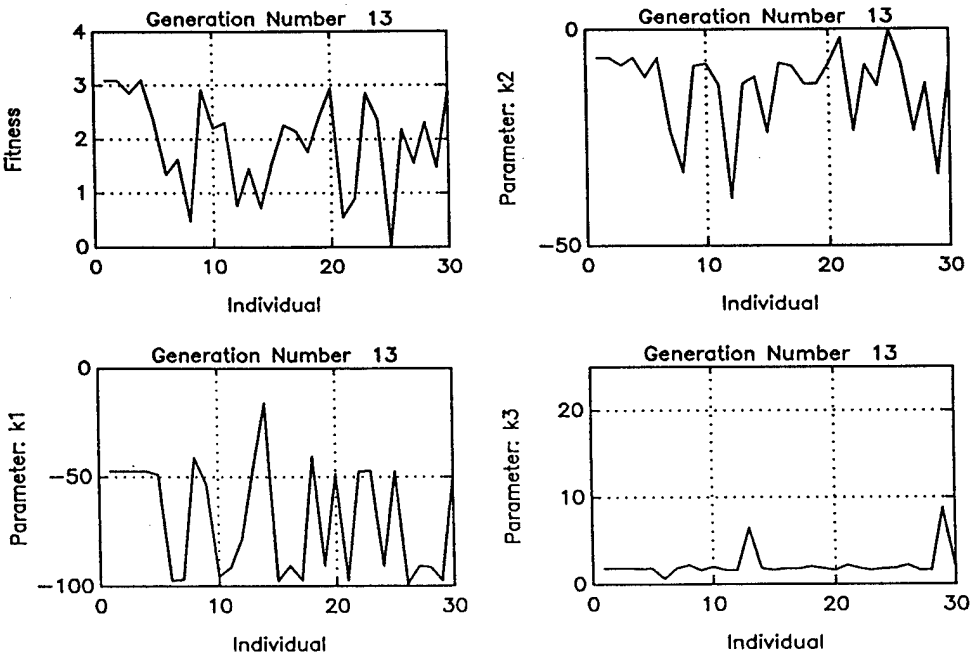

Figure 4.7: Genetic Algorithm Optimization - Generation 13 - J Controller

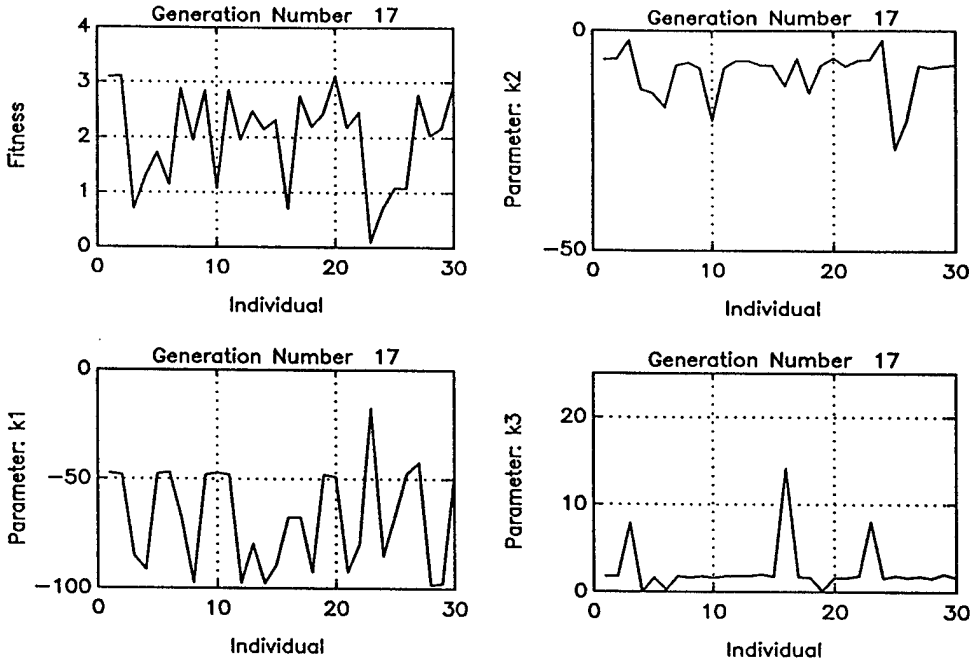

Figure 4.8: Genetic Algorithm Optimization - Generation 17 - J Controller

As noted previously in section 3.4, each parameter word in the genetic algorithm optimization process is scaled separately, so that it can only represent gain values between k_{lower} and k_{upper}. The results of the steady state LQR solution of the previous section provide a good indication of both the magnitude and ***sign*** that should be placed on these limits. Indeed, the resulting closed loop system is stable only if k_1 and k_2 are negative, and k_3 is positive. Thus to accelerate the genetic algorithm convergence, the following upper and lower values were chosen for scaling the parameter words:

parameter	k_{lower}	k_{upper}
k_1	-100.0	0.0
k_2	-50.0	0.0
k_3	0.0	25.0

It should be noted however, that it is not necessary to limit the gain param-

eters to the correct sign values in order for the genetic algorithm to work. Stability, as a *hard constraint*, is contained within the quadratic performance index J. Unstable systems, due to an incorrect feedback gain sign, result in an extremely large value of J, and are appropriately penalized by the reproduction function of the genetic algorithm. In trials where k_{lower} was set to -100 and k_{upper} to +100 for all three gain parameters, the genetic algorithm converged to the stable sign region for all parameters within a few generations.

Several other parameters that are used to control the genetic algorithm process were determined empirically, and were set as follows:

GA Parameters

population size	=	30.000
probability of crossover (p_{cross})	=	.800
probability of mutation ($p_{mutation}$)	=	.025

Figure 4.9 shows the maximum fitness vs. generation in greater detail for the J Controller optimization, and figure 4.10 shows best gain parameters plotted vs. generation. It is seen that the genetic algorithm converges rapidly in this case, and is within a few percent of its final gain values within 3 generations. The feedback controller gains resulting after 17 generations are shown below.

J Controller Gains

k_1 = - 48.9113

k_2 = - 6.2638

k_3 = 1.6400

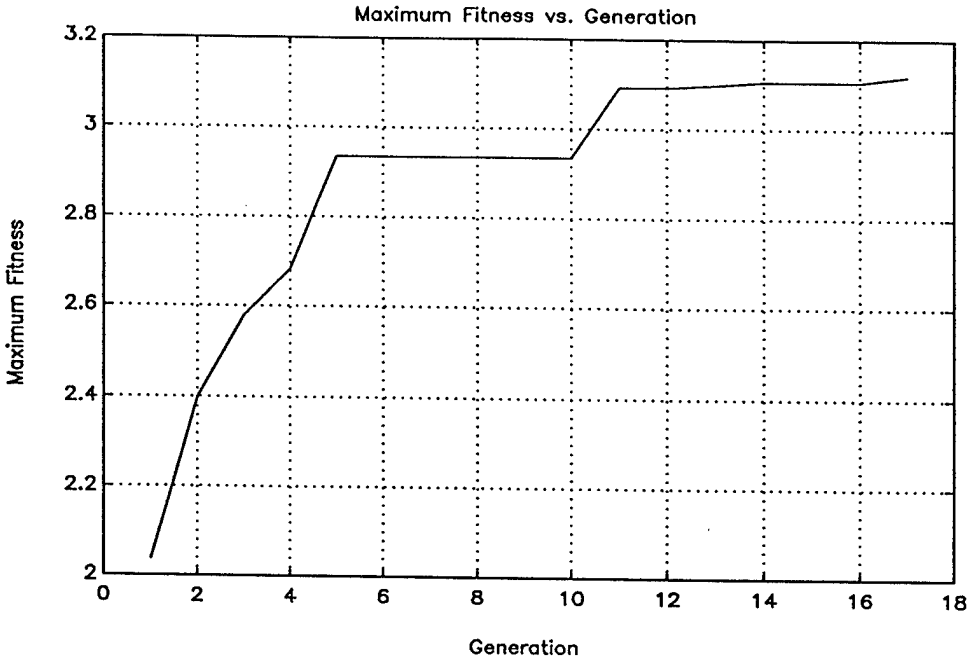

Figure 4.9: Genetic Algorithm - Maximum Fitness - J Controller

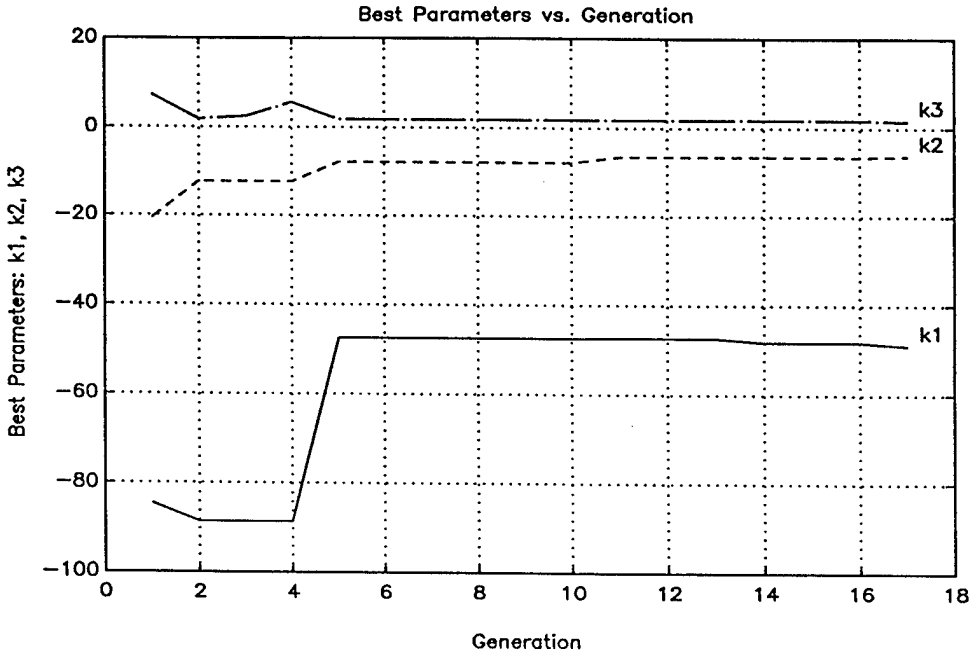

Figure 4.10: Genetic Algorithm - Best Parameters - J Controller

Figures 4.11 and 4.12 show the time response plots for the resulting J Controller. Analysis of this controller design indicates the following characteristics:

J Controller Characteristics

δ_{max}	=	9.7258
ϵ_{settle}	=	.0232
t_{rise}	=	.1970
ϕ_{margin}	=	48.0 deg.
G_{margin}	=	(> 90 dB)

While these characteristics do not match exactly those of the LQR Controller, they are reasonably close considering the differences in implementing the discrete system equation, the finite integration interval, and the numerical integration of the linear quadratic performance index. The J Controller will therefore be used as a baseline for subsequent optimization results which incorporate additional performance constraints.

4.3 Peak Actuator Response Constraint

The system performance characteristics of the J Controller compare favorably with the characteristics of the LQR Controller design, having somewhat greater rise time and peak actuator response, but less settling error. The J Controller also displays adequate robustness characteristics, with gain and phase margins approaching the theoretical values of the LQR Controller. Let us now assume that the actuator system we wish to use has a physical limit of $\pm$ 5 degrees of travel, and impose this as a constraint on the peak actuator reponse. This constraint is incorporated into the genetic algorithm optimization process by setting the actuator peak response specification term:

$$\phi_\delta = 5.0 \tag{4.1}$$

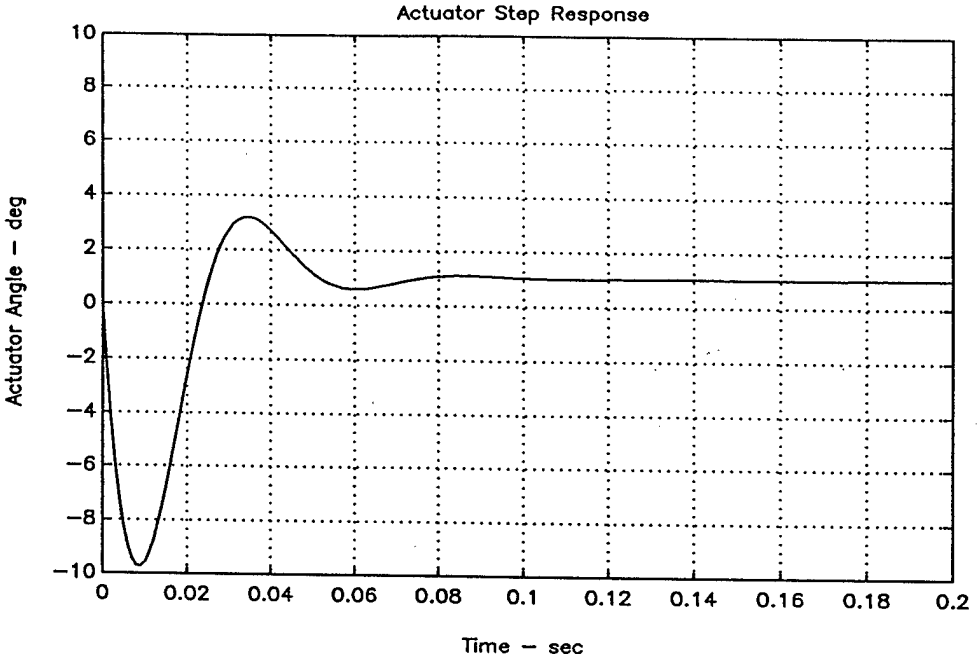

Figure 4.11: Actuator Response for the J Controller

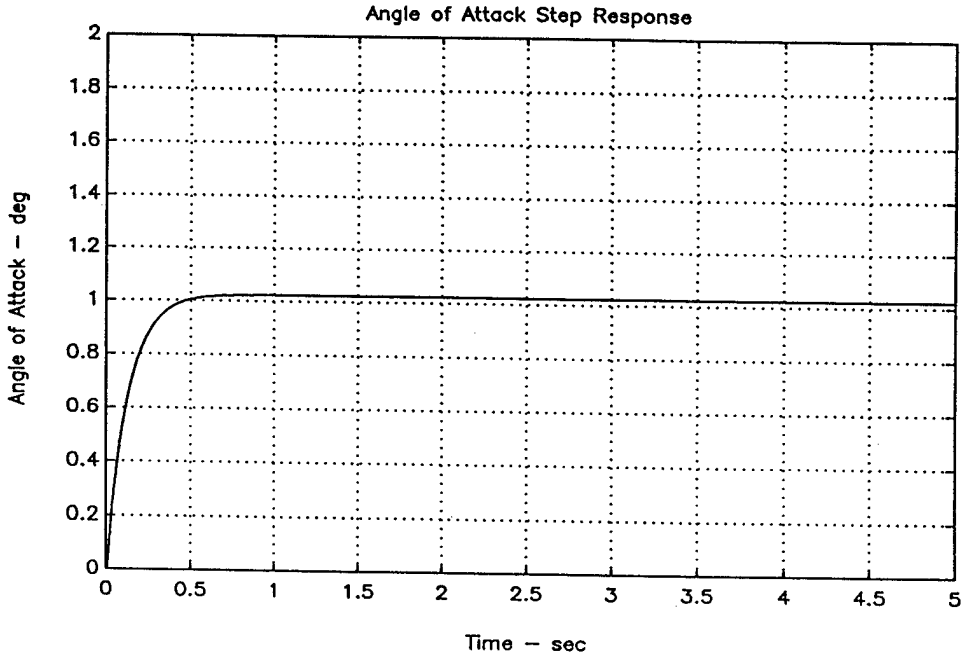

Figure 4.12: Angle of Attack Response for the J Controller

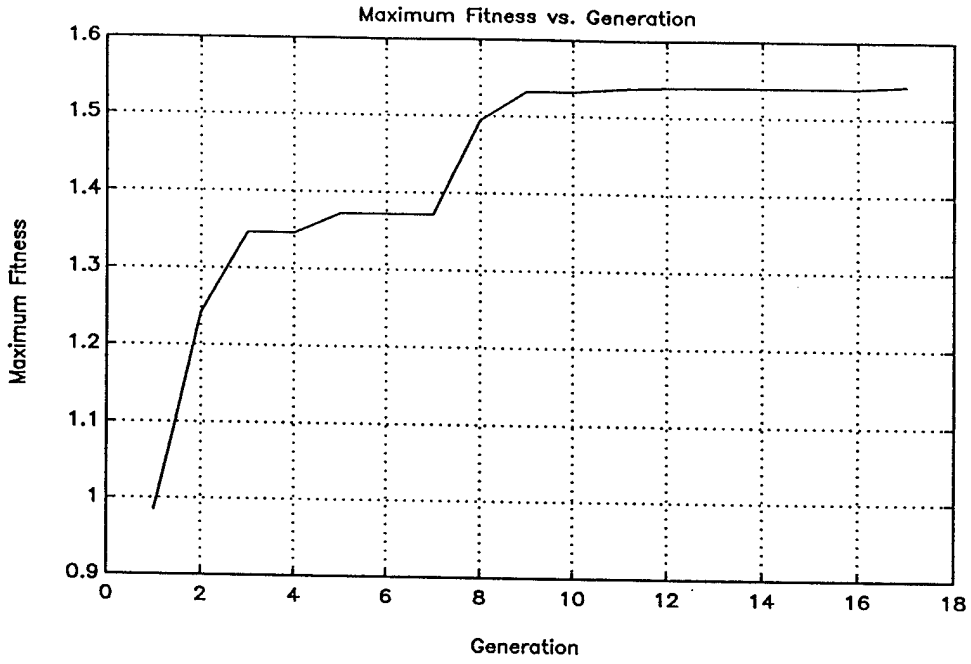

Figure 4.13: Genetic Algorithm - Maximum Fitness - JP_δ Controller

and computing the actuator peak response penalty (P_δ) as defined in equation 3.6. This penalty is then added to the GA cost function, and the resulting genetically optimized system becomes the **JP_δ Controller.**

Figure 4.13 shows the maximum fitness vs. generation in greater detail for the JP_δ Controller optimization, and figure 4.14 shows best gain parameters plotted vs. generation. The feedback controller gains resulting after 17 generations are shown below.

JP_δ Controller Gains

$$k_1 = -66.0487$$

$$k_2 = -14.9935$$

$$k_3 = 9.1844$$

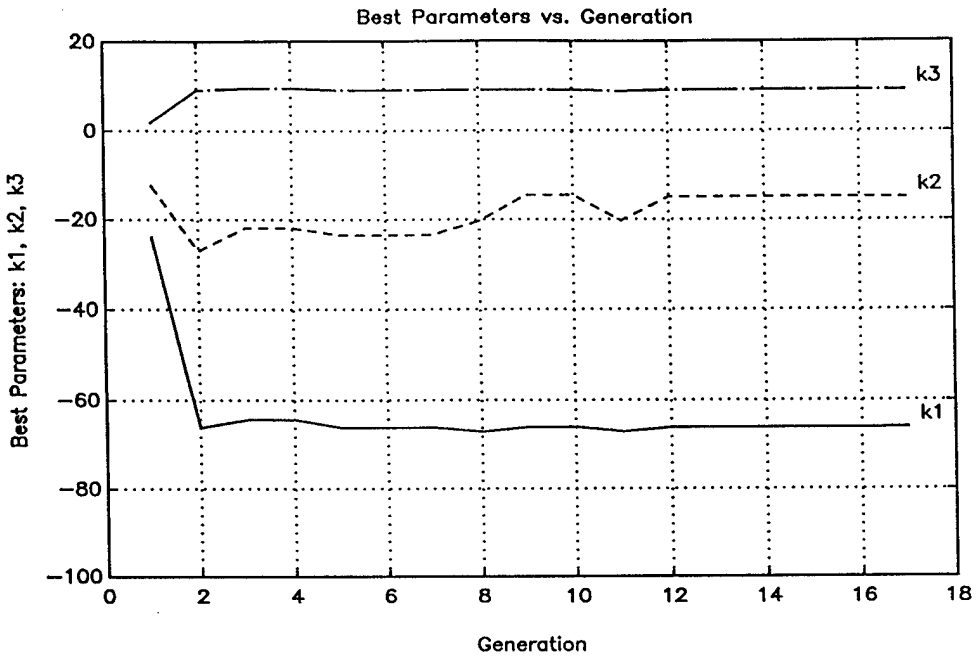

Figure 4.14: Genetic Algorithm - Best Parameters - JP_δ Controller

Figures 4.15 and 4.16 show the time response plots for the JP_δ Controller. Analysis of this controller design indicates the following characteristics:

JP_δ Controller Characteristics

δ_{max}	=	4.9977
ϵ_{settle}	=	.1116
t_{rise}	=	.3200
ϕ_{margin}	=	83.72 deg.
G_{margin}	=	(> 90 dB)

It is observed that the peak actuator response constraint has been met by the new controller design, and the stability margin data show that robustness has not been compromised. However, it is noted that the settling error has increased dramatically with this design. Rising from approximately 2% for the previous controller, the settling error for this design is now over 11% of the commanded value.

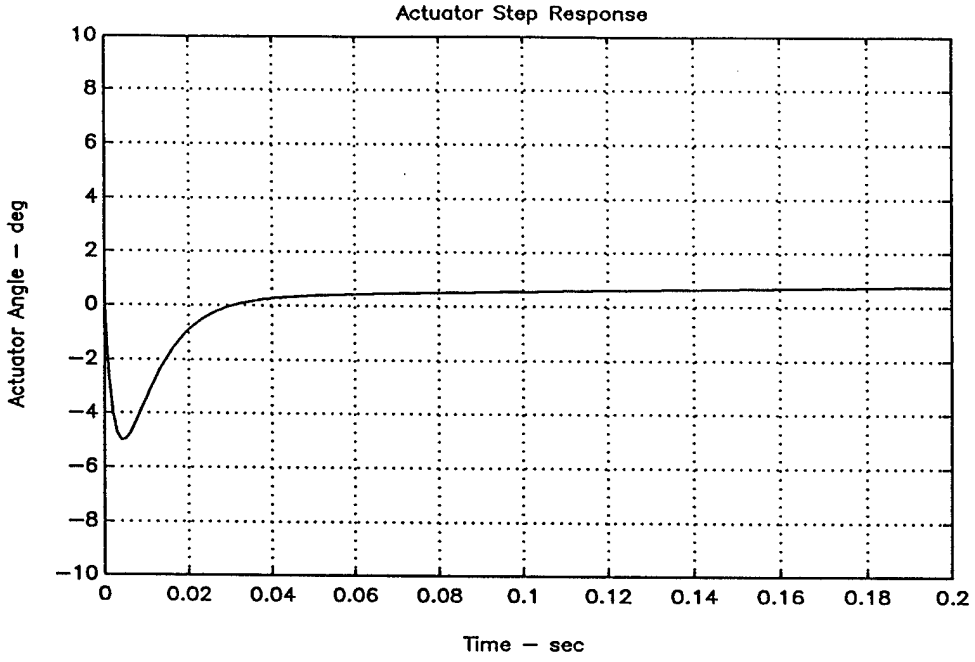

Figure 4.15: Actuator Response for the JP_δ Controller

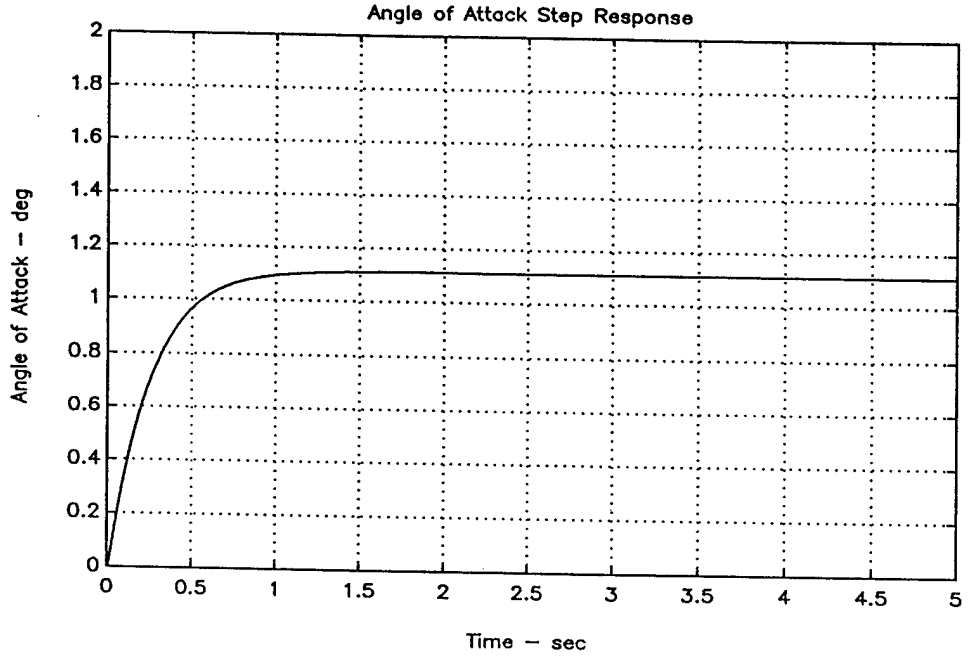

Figure 4.16: Angle of Attack Response for the JP_δ Controller

4.4 Settling Error Constraint

Let us now assume that the settling error observed in the JP_δ Controller is unsatisfactory in terms of system performance, and impose this as an additional constraint on the optimization problem. This constraint is incorporated into the genetic algorithm by adding the weighted settling error penalty (P_{settle}) as defined in equation 3.7 to the GA cost function. The resulting genetically optimized system, which incorporates peak actuator response and settling error constraints, will be referred to as the $JP_\delta P_{settle}$ **Controller.**

Figure 4.17 shows the maximum fitness vs. generation in greater detail for the $JP_\delta P_{settle}$ Controller optimization, and figure 4.18 shows best gain parameters plotted vs. generation. The feedback controller gains resulting after 17 generations are shown below.

$JP_\delta P_{settle}$ Controller Gains

$$k_1 = -24.7425$$
$$k_2 = -7.0481$$
$$k_3 = 1.6289$$

Figures 4.19 and 4.20 show the time response plots for the $JP_\delta P_{settle}$ Controller. Analysis of this controller design indicates the following characteristics:

$JP_\delta P_{settle}$ Controller Characteristics

$$\delta_{max} = 4.7937$$
$$\epsilon_{settle} = .0377$$
$$t_{rise} = .4400$$
$$\phi_{margin} = 46.7 \text{ deg.}$$
$$G_{margin} = (> 90 \text{ dB})$$

It is observed that the settling error has been reduced to less than 4% by the new controller design, and peak actuator response constraint has been simul-

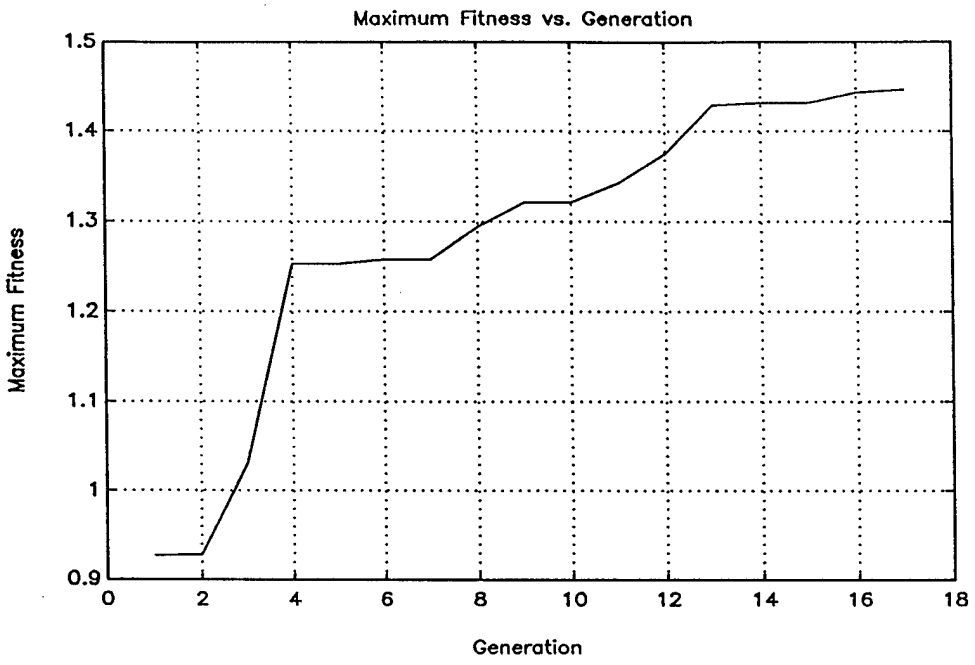

Figure 4.17: Genetic Algorithm - Maximum Fitness - $JP_\delta P_{settle}$ Controller

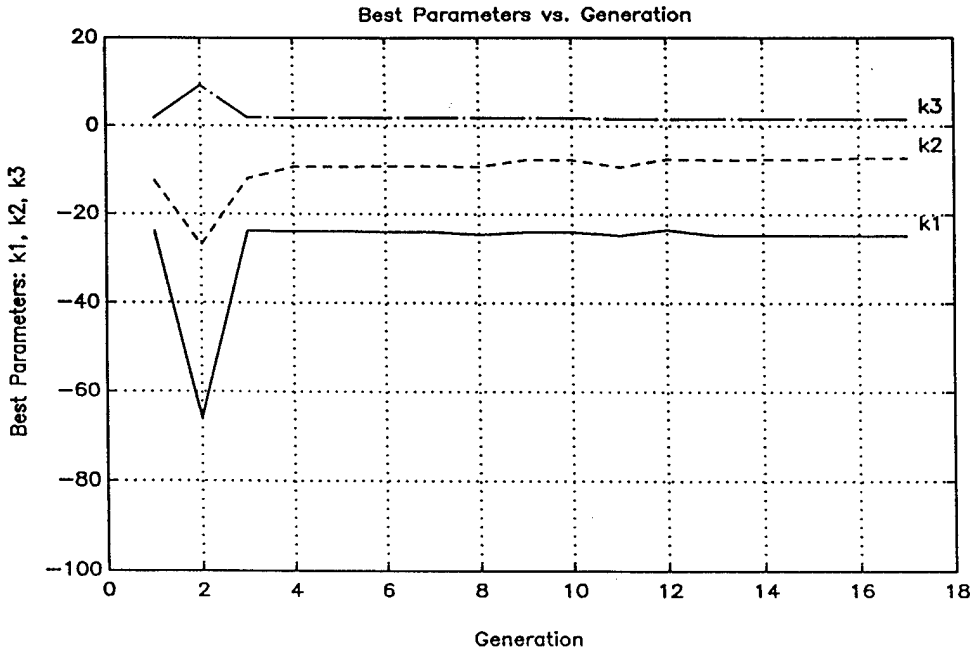

Figure 4.18: Genetic Algorithm - Best Parameters - $JP_\delta P_{settle}$ Controller

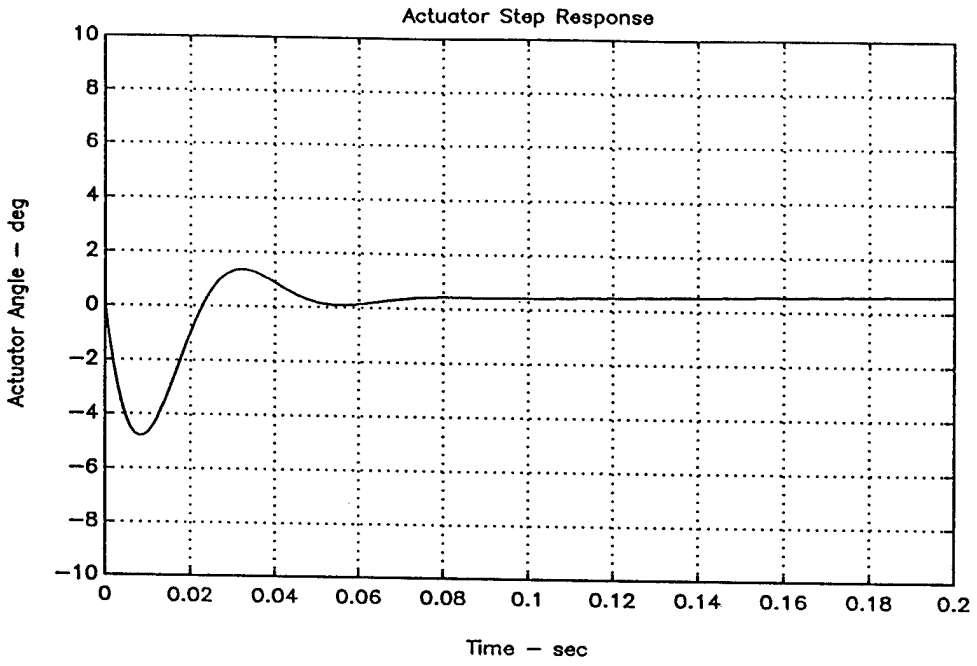

Figure 4.19: Actuator Response for the $JP_\delta P_{settle}$ Controller

taneously satisfied. Stability margin data also indicate that robustness has not been compromised. However, it is noted that the rise time of the step response has increased dramatically from .32 seconds for the previous design to a sluggish .44 seconds for this design.

4.5 Rise Time Constraint

Let us now assume that the sluggish rise time observed in the $JP_\delta P_{settle}$ Controller is unsatisfactory, and that the system performance specification requires that the rise time be no greater that .3 seconds. This final additional constraint is incorporated into the genetic algorithm optimization process, by setting the rise time specification term:

$$\phi_{rise} = .3 \tag{4.2}$$

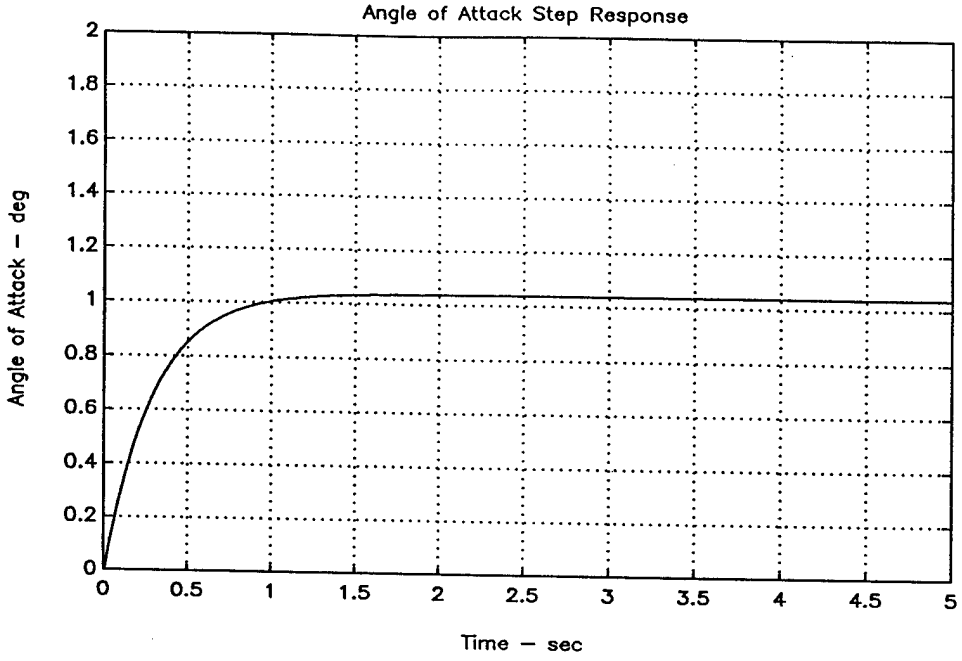

Figure 4.20: Angle of Attack Response for the $JP_\delta P_{settle}$ Controller

and computing the rise time penalty (P_{rise}) as defined in equation 3.5. This penalty is then added to the GA cost function, and the resulting genetically optimized system becomes the $JP_\delta P_{settle} P_{rise}$ **Controller.**

Figure 4.21 shows the maximum fitness vs. generation in greater detail for the $JP_\delta P_{settle} P_{rise}$ Controller optimization, and figure 4.22 shows best gain parameters plotted vs. generation. The feedback controller gains resulting after 17 generations are shown below.

$JP_\delta P_{settle} P_{rise}$ **Controller Gains**

$$k_1 = -22.3987$$

$$k_2 = -4.5464$$

$$k_3 = 1.6514$$

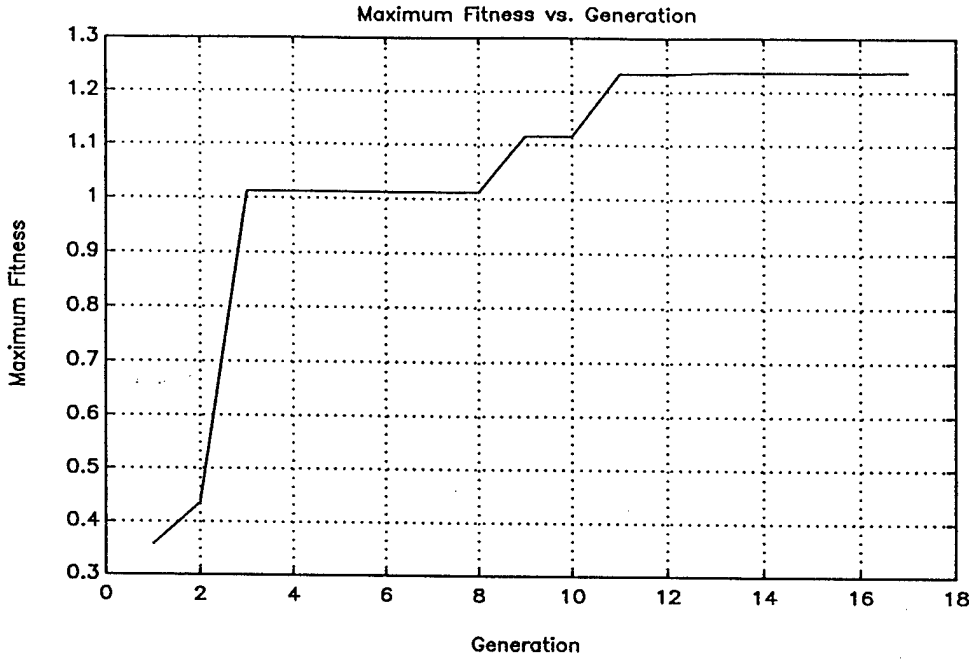

Figure 4.21: Genetic Algorithm - Maximum Fitness - $JP_\delta P_{settle} P_{rise}$ Controller

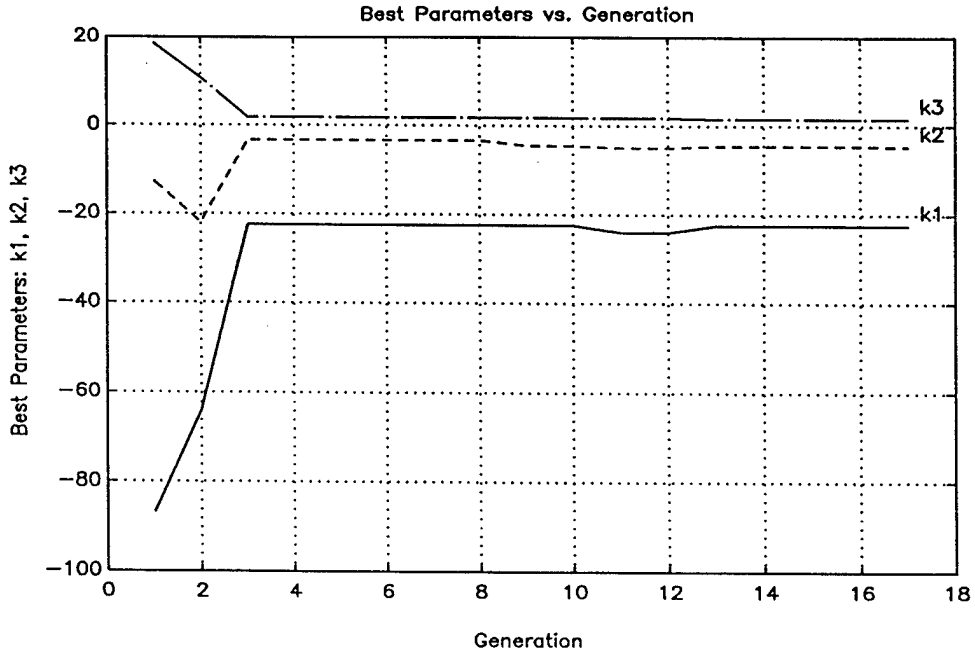

Figure 4.22: Genetic Algorithm - Best Parameters - $JP_\delta P_{settle} P_{rise}$ Controller

Figures 4.23 and 4.24 show the time response plots for the $JP_\delta P_{settle} P_{rise}$ Controller. Analysis of this controller design indicates the following characteristics:

$JP_\delta P_{settle} P_{rise}$ Controller Characteristics

δ_{max}	=	4.8502
ϵ_{settle}	=	.0750
t_{rise}	=	.3000
ϕ_{margin}	=	56.49 deg.
G_{margin}	=	(> 90 dB)

It is observed that the rise time has been reduced to .3 seconds, which meets exactly the system performance specification. Also, the peak actuator response is within its physical limit of $\pm$ 5 degrees of travel. Meanwhile, the settling error has increased to 7.5%, but is still lower than the 11% obtained with no settling error penalty, and may be acceptable. Adequate gain and phase margins are also maintained for robustness of this design.

4.6 Genetic Algorithm Performance

In order to determine the accuracy and performance of the genetic algorithm in this application, the MATLAB program APNMOPT was constructed to find the true optimum solution in each of the controller specifications presented. The APNMOPT program uses the standard MATLAB function, FMINS, which implements the Nelder-Meade simplex algorithm to find the minimum of a multivariable function. Since it is known that this algorithm is not well suited for finding global optimums, the optimum gain values found previously by the genetic algorithm were used as the initial *guess* required by the Nelder-Meade algorithm.

The true optimum gain and fitness function values obtained by the Nelder-Meade algorithm for each of the four controller specifications studie, are summarized

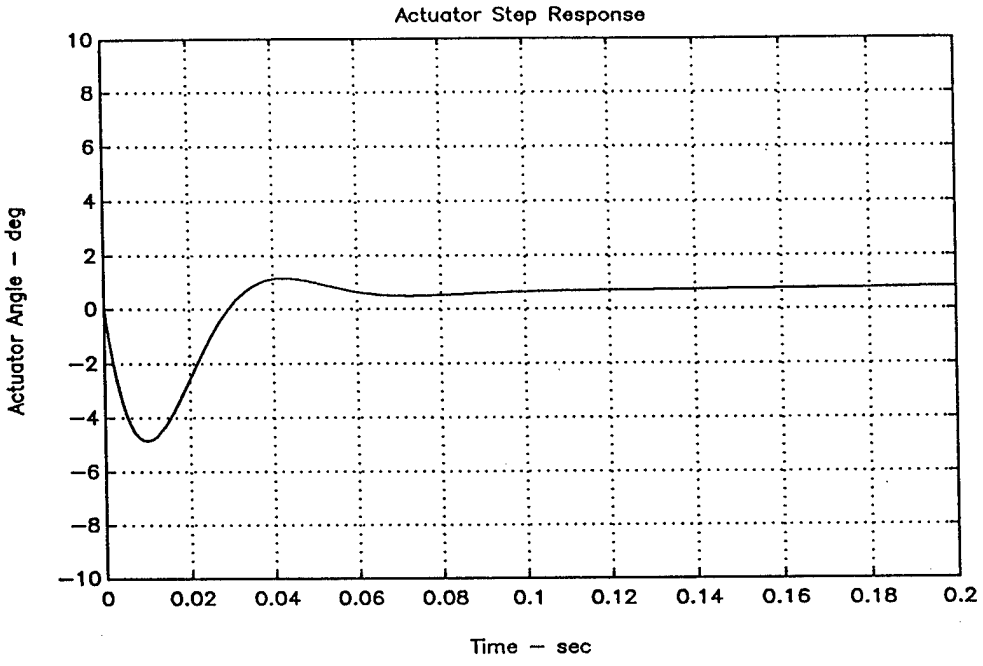

Figure 4.23: Actuator Response for the $JP_\delta P_{settle} P_{rise}$ Controller

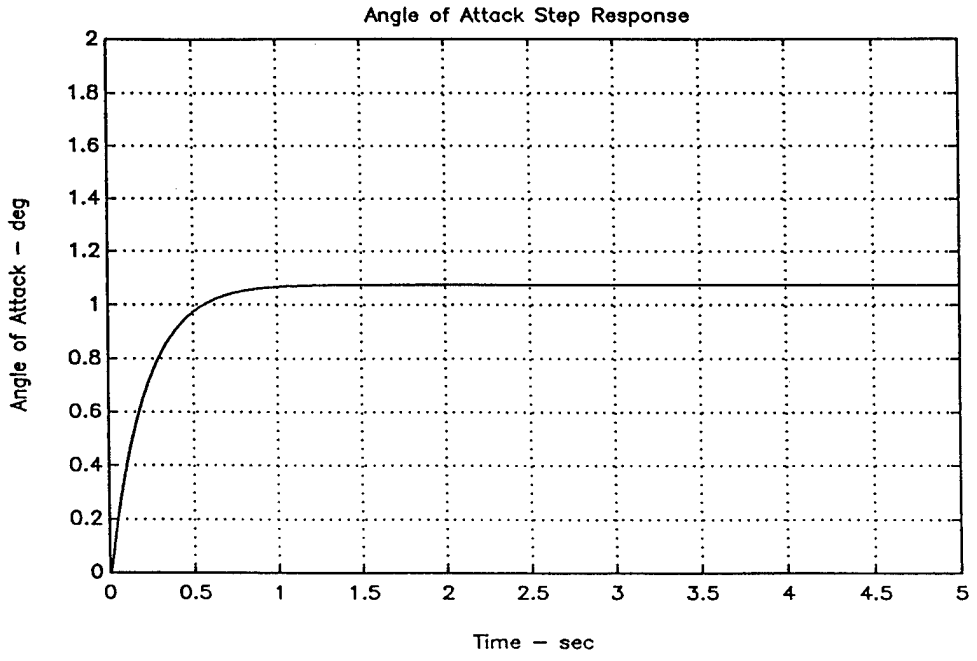

Figure 4.24: Angle of Attack Response for the $JP_\delta P_{settle} P_{rise}$ Controller

in the table below. The genetic algorithm results are also summarized in a similar table for comparison.

Nelder-Meade Optimization Summary

Controller	k_1	k_2	k_3	Fitness
J	-27.0995	-3.5254	0.4122	3.2230
JP_δ	-65.1220	-16.0587	8.8373	1.5807
$JP_\delta P_{settle}$	-21.4221	-6.9169	0.7232	1.7239
$JP_\delta P_{settle} P_{rise}$	-22.2218	-4.4455	1.4819	1.3316

GA Optimization Summary

Controller	k_1	k_2	k_3	Fitness
J	-48.9113	-6.2638	1.6400	3.1183
JP_δ	-66.0487	-14.9935	9.1844	1.5429
$JP_\delta P_{settle}$	-24.7425	-7.0481	1.6289	1.4465
$JP_\delta P_{settle} P_{rise}$	-22.3987	-4.5464	1.6514	1.2367

The resulting errors in the genetic algorithm optimized fitness functions are compared with the *true optimum* values obtained with the Nelder-Meade algorithm in the table below:

Peak Fitness Summary

Controller	GA Optimum	True Optimum	Error
J	3.1183	3.2230	3.2 %
JP_δ	1.5429	1.5807	2.4 %
$JP_\delta P_{settle}$	1.4465	1.7239	16.1 %
$JP_\delta P_{settle} P_{rise}$	1.2367	1.3316	7.1 %

From the tables above, it is observed that the genetic algorithm optimum values are close to the true optimum values, with a few possible exceptions. For

the genetically optimized J Controller, the k_1 gain differed considerably from the true optimum value. Nevertheless, this seemed to have little impact on the fitness function, which was within 3.2% of the true optimum value. For the $JP_\delta P_{settle}$ Controller, the GA fitness function was in error by 16.1%, yet the gains k_1, k_2, and k_3 were reasonably close to their true optimum values.

The performance characteristics of the four genetically optimized controllers have been previously analyzed, and are summarized for reference in the table below. Likewise, the performance characteristics of the controllers using the Nelder-Meade *true optimum* gain values are also tabulated below. From this comparison, it is apparent that the GA optimized controllers, although somewhat suboptimal, have good performance characteristics that reflect the embedded combinations of design criteria.

GA Optimized Controller Characteristics

Controller	δ_{max}	ϵ_{settle}	t_{rise}	ϕ_{margin}
J	9.7258	.0232	.1970	48.0 deg
JP_δ	4.9977	.1116	.3200	83.7 deg
$JP_\delta P_{settle}$	4.7937	.0377	.4400	46.7 deg
$JP_\delta P_{settle} P_{rise}$	4.8502	.0750	.3000	56.5 deg

True Optimum Controller Characteristics

Controller	δ_{max}	ϵ_{settle}	t_{rise}	ϕ_{margin}
J	8.2360	.0207	.2000	32.9 deg
JP_δ	5.0000	.1015	.3500	82.1 deg
$JP_\delta P_{settle}$	4.9156	.0000	.5200	31.6 deg
$JP_\delta P_{settle} P_{rise}$	5.0000	.0679	.3000	53.73 deg

For better understanding of the sensitivity of the fitness function to the independent parameters, three comparison plots have been made for each controller specification. Since the resulting four dimensional *performance surfaces* are difficult to visualize and impossible to represent in two dimensions, the following procedure was used:

1. Evaluate and plot the fitness function vs. k_1, while holding constant the GA optimum values for k_2 and k_3.

2. Evaluate and overlay on this plot the fitness function vs k_1, for constant true optimum values of k_2 and k_3.

3. Repeat 1. and 2. above, but generate plots vs. k_2, while using constant values of k_1 and k_3.

4. Repeat 1. and 2. above, but generate plots vs. k_3, while using constant values of k_1 and k_2.

The resulting parametric fitness function comparisons are shown in figures 4.25 through 4.27 for the J Controller, figures 4.28 through 4.30 for the JP_δ Controller, figures 4.31 through 4.33 for the $JP_\delta P_{settle}$ Controller, and figures 4.34 through 4.36 for the $JP_\delta P_{settle} P_{rise}$ Controller. These figures were generated using MATLAB program APOPLOT after the GA and true optimum parameters were known. The (*) on each plot marks the peak value for the true optimum of the fitness function found by the Nelder-Meade algorithm. The (X) on each plot marks the optimum fitness value found by the genetic algorithm for that case.

As indicated by these results, the genetic algorithm converged to near optimal controller designs, within 17 generations, for each progressively more difficult combination of design constraints. It should be noted that the 17 generations of

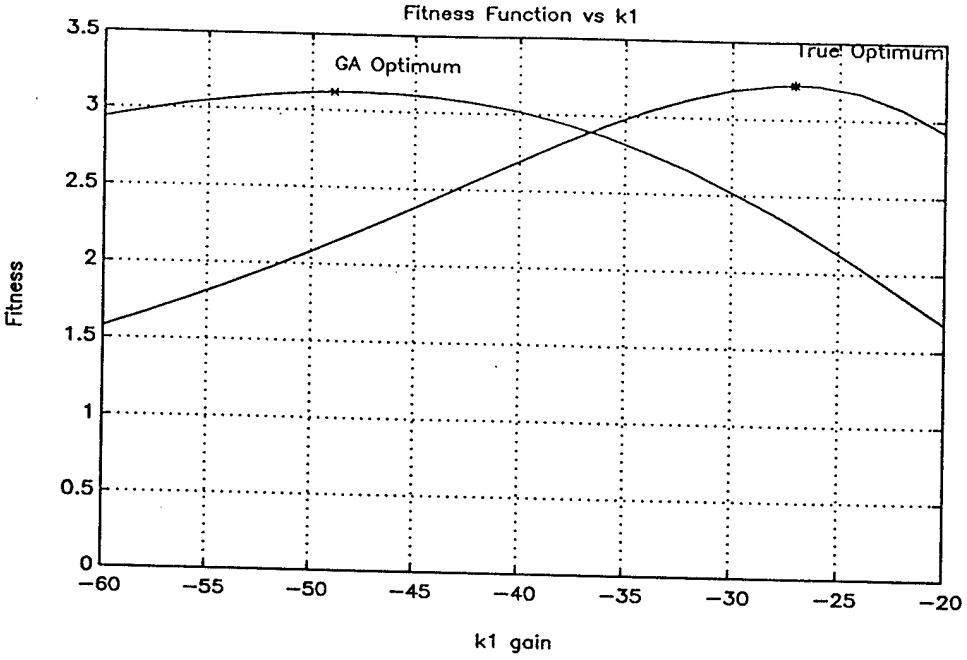

Figure 4.25: Fitness Function vs k_1 for the J Controller Specification

30 individuals represent only 478 combinations (or less) of the three gain parameters (allowing for the fact that the two best individuals are duplicated exactly in each following generation). For practical applications, however, more than 17 generations may be required to assure accurate convergence of the genetic algorithm. The previously cited example by Goldberg required approximately 65 generations to converge [8]. Perhaps a useful strategy would be to use the genetic algorithm for some number of generations to assure global optimality, then switch to a more orthodox numerical search method, such as the Nelder-Meade algorithm, to converge on the final optimum.

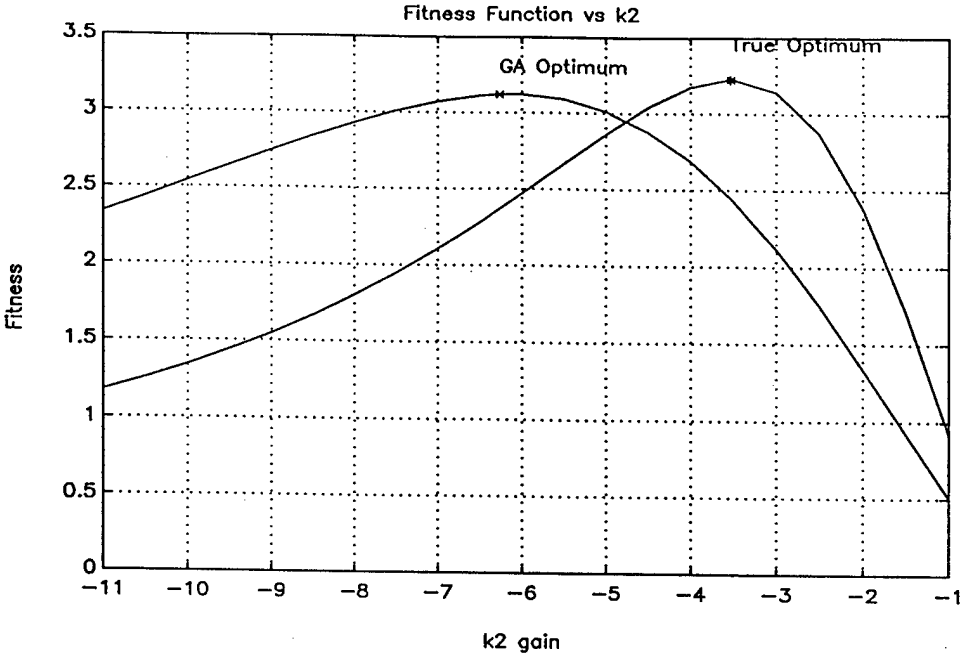

Figure 4.26: Fitness Function vs k_2 for the J Controller Specification

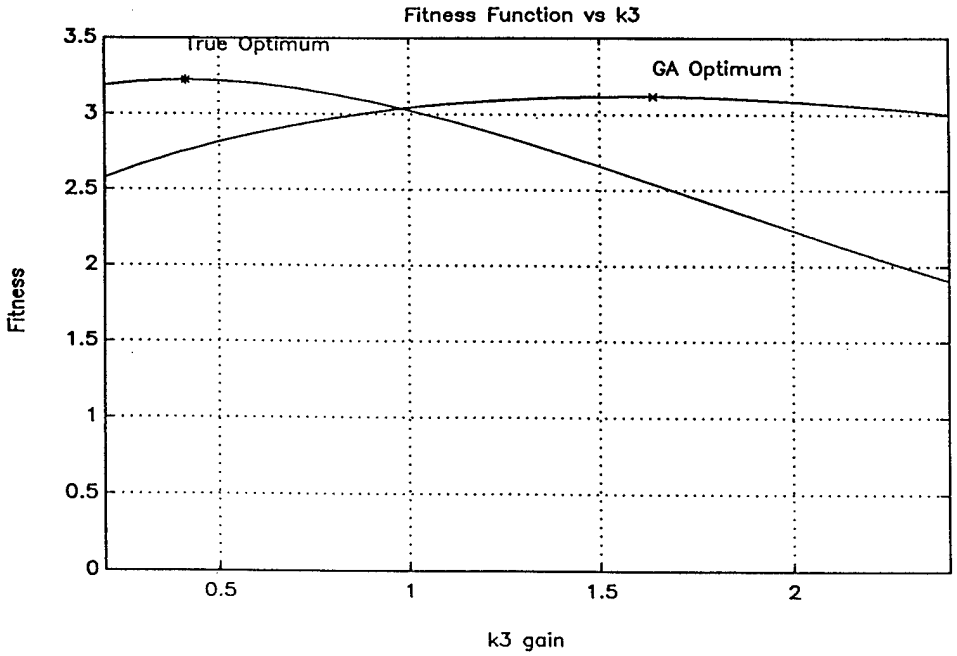

Figure 4.27: Fitness Function vs k_3 for the J Controller Specification

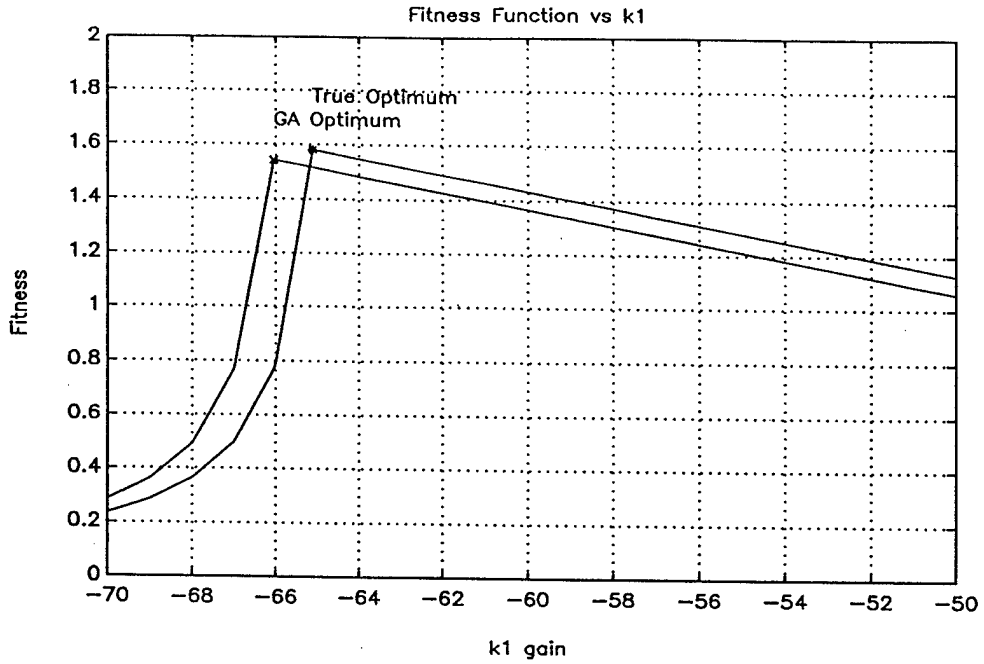

Figure 4.28: Fitness Function vs k_1 for the JP_δ Controller Specification

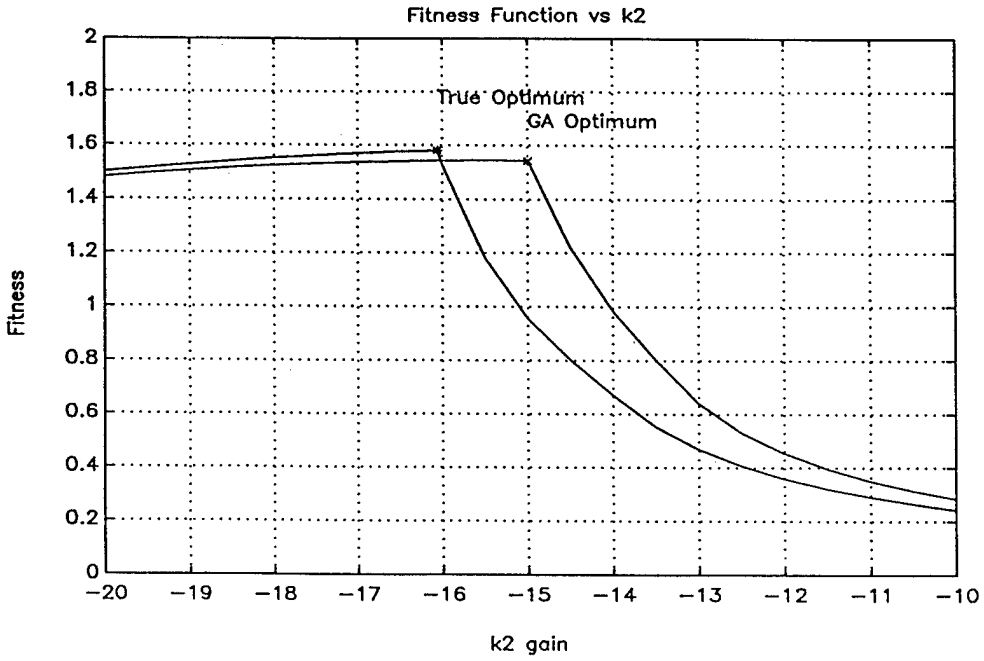

Figure 4.29: Fitness Function vs k_2 for the JP_δ Controller Specification

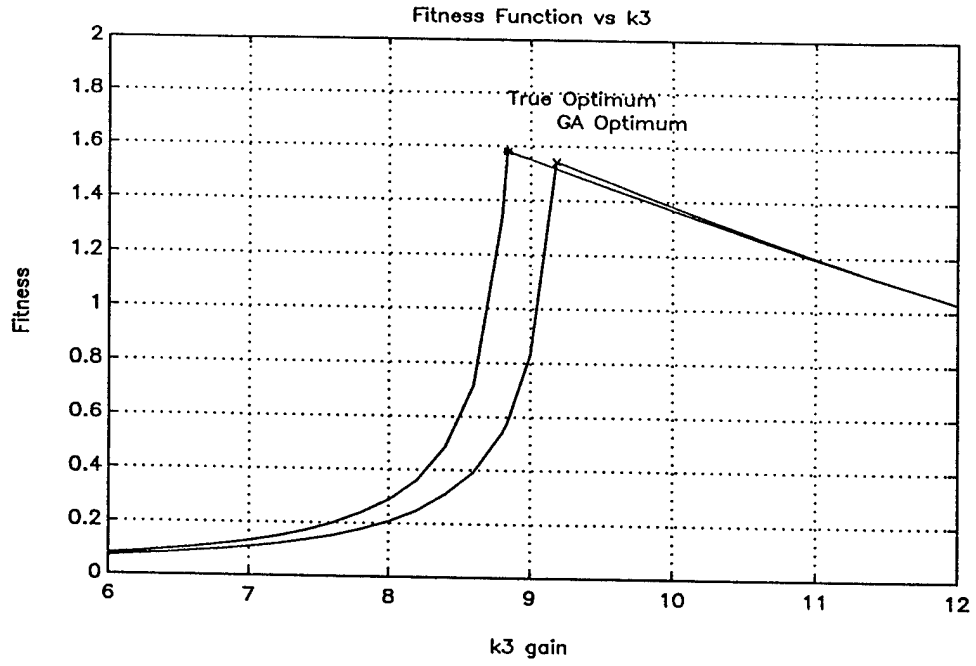

Figure 4.30: Fitness Function vs k_3 for the JP_δ Controller Specification

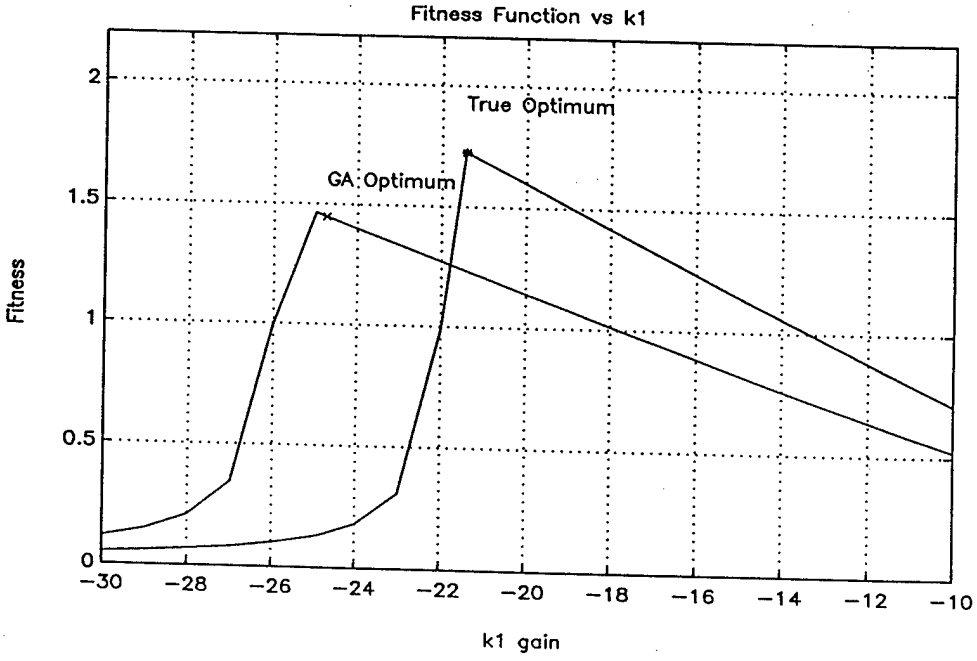

Figure 4.31: Fitness Function vs k_1 for the $JP_\delta P_{settle}$ Controller Specification

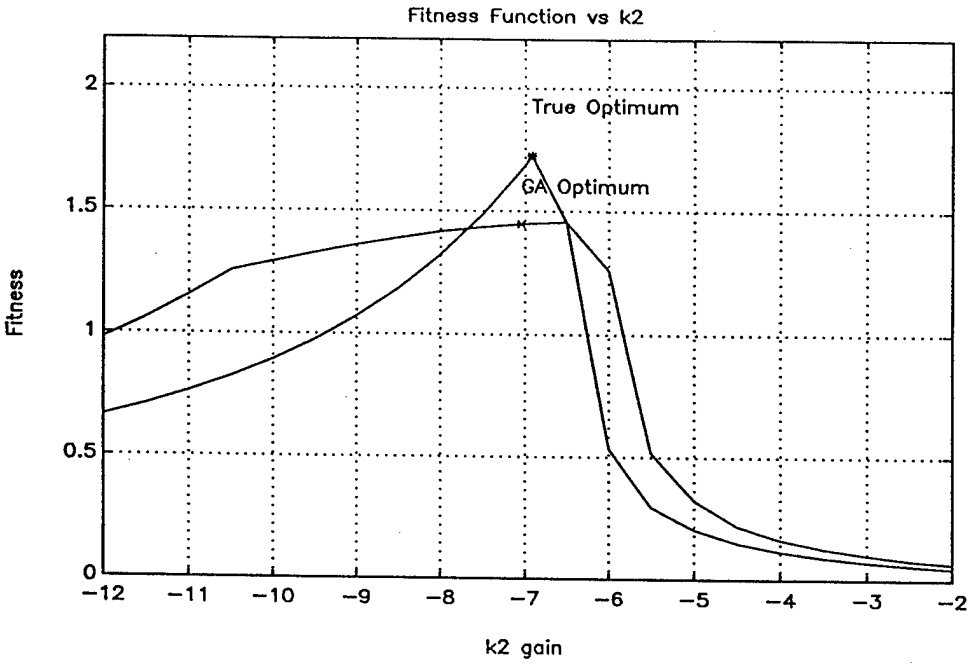

Figure 4.32: Fitness Function vs k_2 for the $JP_\delta P_{settle}$ Controller Specification

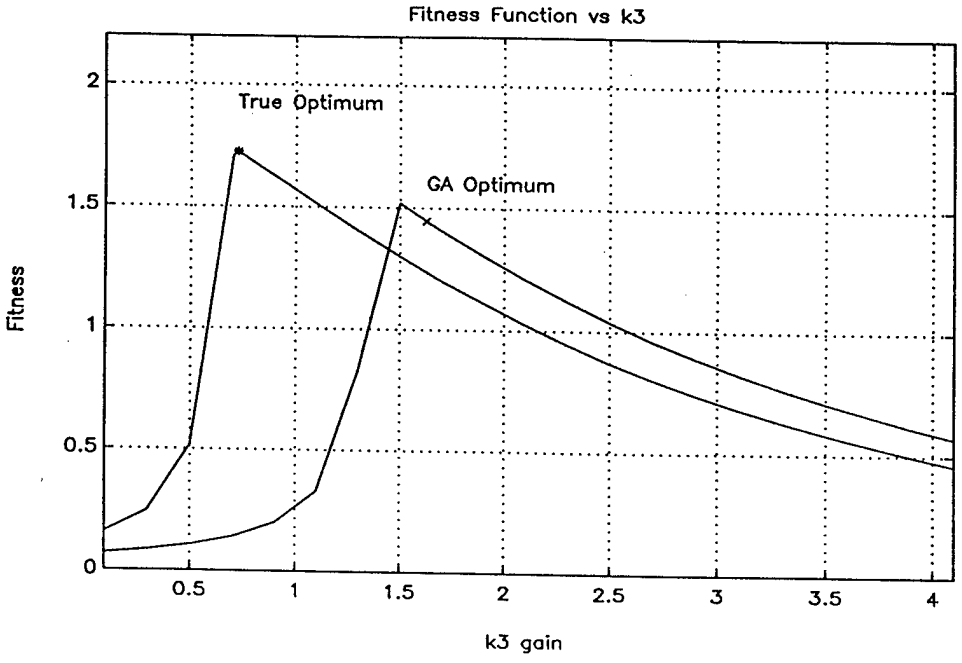

Figure 4.33: Fitness Function vs k_3 for the $JP_\delta P_{settle}$ Controller Specification

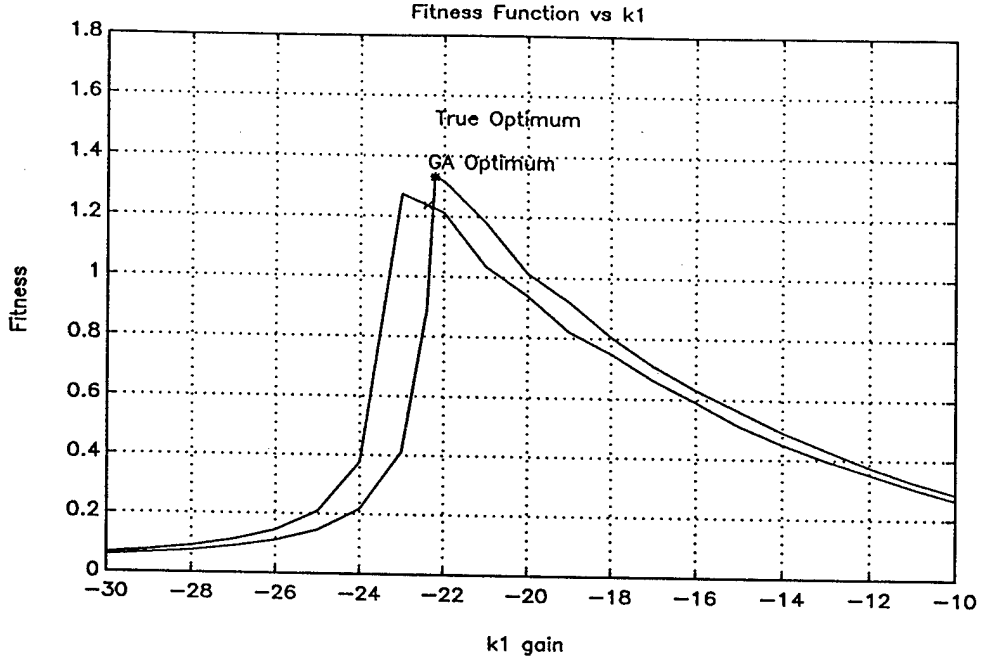

Figure 4.34: Fitness Function vs k_1 for the $JP_\delta P_{settle} P_{rise}$ Controller Specification

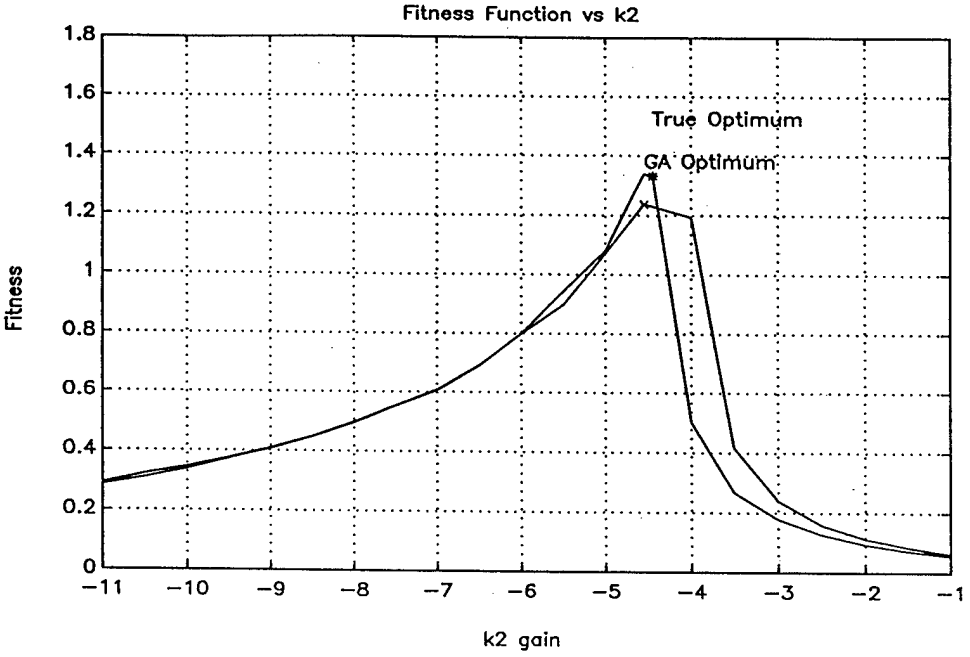

Figure 4.35: Fitness Function vs k_2 for the $JP_\delta P_{settle} P_{rise}$ Controller Specification

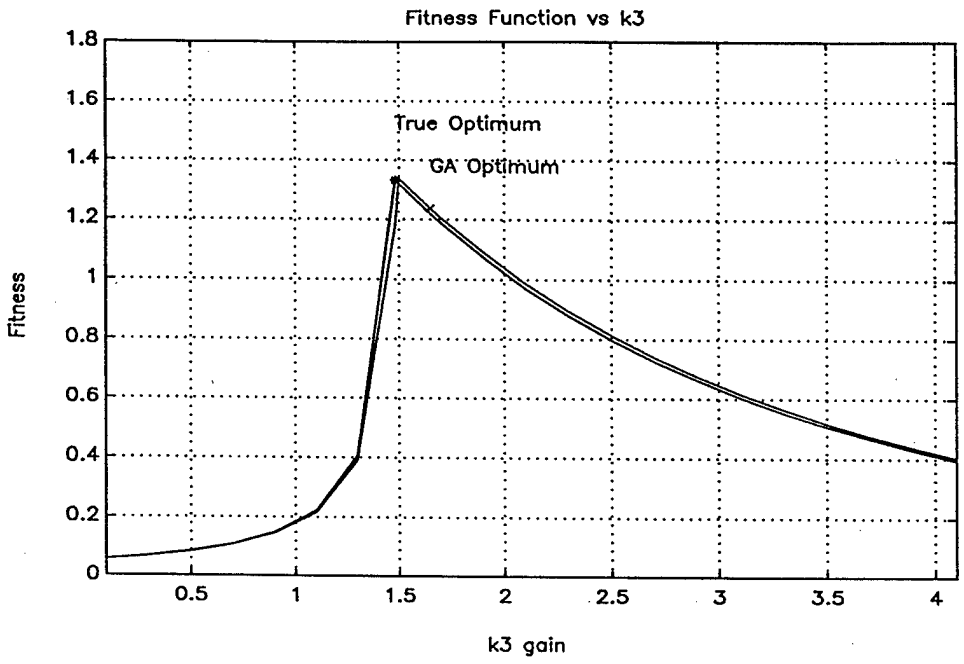

Figure 4.36: Fitness Function vs k_3 for the $JP_\delta P_{settle} P_{rise}$ Controller Specification

CHAPTER 5

CONCLUSIONS

In this thesis a control system optimization problem was formulated with respect to a simple pitch plane missile autopilot. Linearized missile airframe and actuator dynamics models were developed, and a linear state feedback controller was assumed. A set of hypothetical flight parameters were assumed, and a discrete time state space simulation model for the closed loop system was developed.

A discretized linear quadratic performance measure was developed, which can be computed numerically from the closed loop simulation model. Additional performance specifications were then imposed to provide constraints on peak actuator response, settling error, and rise time performance. It was posited that the resulting multicriterion optimization problem was one for which analytic solutions are not available.

The genetic algorithm was then introduced, and a suitable fitness measure was developed that could incorporate the stated performance measure and auxillary constraints. Computer simulation results were then presented that demonstrate the genetic algorithm provides good convergence to near optimal controller designs for successive combinations of constraints. The final genetically optimized controller was designed to minimize a linear quadratic performance index, while simultaneously minimizing settling error and satisfying hard constraints on peak actuator

response and minimum rise time performance. The demonstrated success of the genetic algorithm in this application satisfies the objective of this thesis.

5.1 Future Applications

From this vantage point, future applications for the genetic algorithm in control system optimization seem promising and virtually unlimited. A simple application system model was used in this thesis primarily to meet time and computer resource constraints. Given sufficient processing resources, however, the closed loop system model evaluated by the genetic algorithm for this application could be replaced by a full scale simulation of arbitrary complexity.

Further investigations into suitable performance measures and the incorporation of many additional design contraints are also warranted. Robustness contraints were not addressed directly in this thesis; rather robust stability margins were achieved as a property of full state feedback and application of the linear quadratic performance index. A viable area of study would be to incorporate multivariable robust control specifications into the genetic algorithm fitness function.

The linear feedback controller form used in this thesis was also selected as a comfortable starting place. Most real world control system design problems do not enjoy the luxury of full state feedback, free of sensor noise and time delay. Application of the genetic algorithm optimization approach could easily be applied to other forms of controllers. Realistic control systems applications involving reduced order observers and optimal stochastic observers should be investigated. The GA could potentially be used to optimize both controller and observer in such configurations.

Finally, there is no reason to limit the genetic algorithm optimization process to linear systems, or even to time invariant systems. For time varying systems, it

may one day be possible to implement a genetic algorithm optimization loop in real time. This depends of course upon such things as the processing power available, the complexity of the problem, and the real time available for the optimization loop. It may be seen however, that the genetic algorithm has an inherent advantage for real time implementation within a parallel processing environment. This natural parallelism of the genetic algorithm can be exploited, for example, by invoking simultaneous evaluation of the fitness function for all the individuals (32, or 64, or 128, ...) in a given generation, each assigned to a separate processor. These evaluations are the most time consuming operations of the genetic algorithm optimization process. The serial operations required to select and mate individuals between generations represent only a small fraction of the total processing time requirements!

Not every controller optimization problem is feasible, but for those that are feasible, the genetic algorithm can provide a useful tool for automating the controller design process.

Bibliography

[1] Brian D. O. Anderson and John B. Moore, **Optimal Control, Linear Quadratic Methods**, Prentice Hall, 1990.

[2] Stephen P. Boyd and Craig H. Barratt, **Linear Controller Design, Limits of Performance**, Prentice Hall, 1991.

[3] Robert Grover Brown and Patrick Y. C. Hwang, **Introduction to Random Signals and Applied Kalman Filtering**, Second Edition, John Wiley & Sons, Inc., 1992.

[4] Richard Y. Chiang and Michael G. Safonov, **Robust-Control Toolbox for use with MATLAB**, The Math Works, Inc., Prentice Hall, 1988.

[5] Bernard Friedland, **Control System Design**, McGraw Hill, 1986.

[6] Arthur Gelb, editor, **Applied Optimal Estimation**, The M.I.T. Press, 1974.

[7] David E. Goldberg, **Genetic Algorithms in Search, Optimization & Machine Learning**, Addison Wesley, 1989.

[8] K. Krishna Kumar and David E. Goldberg, *Genetic Algorithms in Control System Optimization*, AIAA, 1990.

[9] Donald E. Kirk, **Optimal Control Theory, An Introduction**, Prentice Hall, 1970.

[10] John N. Little and Alan J. Laub, **Control Toolbox for use with MATLAB**, The Math Works, Inc., Prentice Hall, 1988.

[11] J. M. Maciejowski, **Multivariable Feedback Design**, Addison Wesley, 1989.

[12] John H. Mathews, **Numerical Methods for Mathematics, Science and Engineering**, Second Edition, Prentice Hall, 1992.

[13] The Math Works, Inc., **The Student Edition of MATLAB** , Prentice Hall, 1992.

[14] Charles L. Phillips and Royce D. Harbor, **Feedback Control Systems**, Second Edition, Prentice Hall, 1991.

[15] Richard Wavell, *Current Issues in Real-Time Simulation*, published class notes for ECM 6428, University of Central Florida, Summer Semester, 1990.

APPENDIX A

MATLAB Programs

Several programs were written in MATLAB to produce the analysis and generate the various plots presented in this thesis. This appendix briefly describes the main programs that were used and provides a listing of these programs. The supporting functions that were created in MATLAB especially for these programs are described in the following appendix. In addition these programs require numerous standard MATLAB functions, as well as supporting functions from the MATLAB Control System Toolbox [13,10].

A.1 APGEN3

This MATLAB program implements a genetic algorithm in order to optimize a missile linear feedback controller. It provides continuous on screen plots demonstrating algorithm progress, and provides automatic or manually controlled options to send output to a printer. Several experimental options are controlled by use of option flags, as explained in the comments.

A listing of this program follows.

```
echo on
clc
%   APGEN3  - Autopilot Genetic Algorithm Design Program 2
%           - optimize three parameters k1 & k2 & k3
%           - airframe model only
%           - with linear feedback controller
%           - numerical cost function computation
%             for optimization studies
%           - using LQR cost function
%           - EVALPOP3 includes rise time penalty ( 1/26/93 )
%           - corrected N_delta ( 2/8/93 )
%           - Actuator model ( 2/8/93 )
%           - EVALPOP4 to include hang off error penalty (2/8/93)
%           - EVALPOP5 to use 3 parameterr ( 2/9/93 )
%           - Limit squeeze algorithm  ( 2/10/93 )
%           - Option to vary crossover site distribution with
%             generation number  ( 2/15/93 )
%           - EVALPOP6 to include actuator max response penalty
%             ( 2/23/93 )
%           - Add hoe_thresh to EVALPOP6  ( 2/24/93 )
%           - EVALPOP7 and STEP2S to use split dt's for evaluating
%             step function - to correct actuator norm ( 3/1/93 )
%
%             by    R. Hull,    2/9/93
%
%  This program implements a genetic algorithm to optimize the
%    autopilot controller for a simplified missile autopilot system.
%
%
%    Special Constants :

     HztR = 2*pi;          % Converts Hz to Radians per Second

     walsh_string = [ 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
                      1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
                      1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0
                      1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
                      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                      0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
                      0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
                      0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
                      0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
                      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]

%    Global Variables :

     global cont_plot ;

     cont_plot = 1 ;        %   Continuous plot flag

     auto_plot = 1 ;        %   Automatic plot flag ( to printer )

     combine_flag = 0 ;     %  Option to combine old and new populations
                            %    and save the best individuals of both.
```

```
xover_flag = 0 ;        %  Use crossover site distribution as a
                        %  function of generation number.

squeeze_lim = 0 ;       %  Squeeze limits flag

%    Genetic Algorithm Structures

maxgen = 17 ;           %  total number of generations

print_gen = 4 ;         %  number of generations per output loop

popsize = 30 ;          % number of individules in population
                        %   - must be even !

topsize = 2 ;           % number (even) of top ranked individuals to
                        %  propagate exactly in the next generation

if combine_flag == 1    % Make sure topsize is 0 if combine option
  topsize = 0 ;         %   is used.
end ;

newsize = 0 ;           % number ( even ) of new random individuals
                        %  to include in each new generation

numparams = 3 ;         % number of parameters to optimize

paramlength = 16 ;  % number of bits per parameter

stringlength = numparams * paramlength ;    % genetic string length
                                            %  ( bits )

dt = .01 ;              % Computaion interval - sec.

t_final = 5.0 ;         % cost & fitness function final time - sec.

dts = .001 ;            % Small computaion interval - sec.

ts_final = 0.2 ;        % Small dt response final time - sec.

dtl = .01 ;             % Small computaion interval - sec.

tl_final = 5.0 ;        % Small dt response final time - sec.


% J Controller Specification :
    hoe_thresh = 100.0 ; % settling error penalty threshold
    phi_rise =   100.0 ; % rise time penalty threshold
    act_thresh = 100.0 ; % maximum actuator response specification
                         %  penalty threshold

% J Pd  Controller Specification :
    hoe_thresh = 100.0 ; % settling error penalty threshold
    phi_rise =   100.0 ; % rise time penalty threshold
    act_thresh = 5.0 ;   % maximum actuator response specification
                         %  penalty threshold
```

```
% J Pd Psettle  Controller Specification :
    hoe_thresh = 0.0 ;   % settling error penalty threshold
    phi_rise =  100.0 ;  % rise time penalty threshold
    act_thresh = 5.0 ;   % maximum actuator response specification
                         %  penalty threshold


% J Pd Psettle Prise Controller Specification :
    hoe_thresh = 0.0     % settling error penalty threshold
    phi_rise =    .30    % rise time penalty threshold
    act_thresh = 5.0     % maximum actuator response specification
                         %  penalty threshold

 % Verify controller specification
    pause

    pcross = .8 ;          % probability of crossover

    pmutation = .025       % probability of mutation

    kg_scale = [ -100  0
                  -50  0
                    0  25 ] ;  % gain parameter scales

    init_scale = kg_scale     % used for limit squeeze algorithm

    squeeze_factor = 0.9     % used for limit squeeze algorithm (< 1.0)

    squeeze_buffer = 2.0     % used for limit squeeze algorithm ( > 0 )


%    Flight Condition Constants :

    M_alpha = 64.11;
    M_delta = -62.34;
    M_thetadot = 0;
    N_alpha = .1803;
    N_delta = .0738;
    Tau_act = .02 ;          % Actuator first order time constant
    Velocity = 892.0;        % meters/sec

% Units Conversions


% Define Airframe + First Order Actuator State Space Realization :

    A = [ - N_alpha  1.0  - N_delta
            M_alpha   0     M_delta
              0       0   - ( 1 / Tau_act ) ] ;

    B = [    0
             0
           ( 1 / Tau_act ) ]
```

```
C = [ ( N_alpha * Velocity )  0   ( N_delta * Velocity ) ] ;

D = [ 0 ] ;

% Determine LQR design for basic plant with first order actuator :

qmatrix = [  10  0  0
             0 .1  0
             0  0 .01  ];

rmatrix = .01;

% Initialize time values

t = 0:dt:t_final ;

ts = 0:dts:ts_final ;

tl = ts_final:dtl:tl_final ;

% Set screen display format to suppress excessive line feeds

format compact ;

% Experimental Option to Initialize Population using Walsh Strings

%  new_chrom = [ ] ;

%  for iw = 1 : popsize

%  new_chrom = [ new_chrom ; ..
%   walsh_string( iw, : ) walsh_string( iw, : ) walsh_string( iw, : ) ]

%    pause

%  end ;

% Initialize New Population to Random Strings

    new_chrom = fix( 1.9999999 * rand( popsize, stringlength ) ) ;

 %  Clear generational statistics

 gen_avg_fitness = [ ] ;

 gen_max_fitness = [ ] ;

 gen_max_param = [ ] ;
```

```
gen_max_chrom = [ ] ;

gen_min_fitness = [ ] ;

gen_min_param = [ ] ;

gen_min_chrom = [ ] ;

gen_kg_lower = [ ] ;

gen_kg_upper = [ ] ;

gen = 1 ;     % generation number

% Evaluate Fitness of Initial Population

[ new_fitness, new_param ] = evalpop7( A, B, C, D, ts, dts, tl, dtl, ..
     hoe_thresh, phi_rise, act_thresh, popsize, kg_scale, new_chrom )
;

% Evaluate Initial Population Statistics

avg_fitness = sum( new_fitness ) / popsize ;

[ max_fitness, imax ] = max( new_fitness ) ;

max_param = new_param( imax, : ) ;

max_chrom = new_chrom( imax, : ) ;

[ min_fitness, imin ] = min( new_fitness ) ;

min_param = new_param( imin, : ) ;

min_chrom = new_chrom( imin, : ) ;

% Save initial generation data :

gen_avg_fitness = [ gen_avg_fitness ; avg_fitness ] ;

gen_max_fitness = [ gen_max_fitness ; max_fitness ] ;

gen_max_param = [ gen_max_param ; max_param ] ;

gen_max_chrom = [ gen_max_chrom ; max_chrom ] ;

gen_min_fitness = [ gen_min_fitness ; min_fitness ] ;

gen_min_param = [ gen_min_param ; min_param ] ;

gen_min_chrom = [ gen_min_chrom ; min_chrom ] ;

gen_kg_lower = [ gen_kg_lower ; kg_scale( :, 1 ).' ] ;
```

```
gen_kg_upper = [ gen_kg_upper ; kg_scale( :, 2 ).' ] ;

    % Display or plot values to screen

    if cont_plot == 1

      clg ;

      % Plot of fitness vs individual for this generation :
      subplot(221) ;
      ind_num = 1 : popsize ;
      plot( ind_num, new_fitness ),
                    title(['Generation Number  ', int2str(gen)] ),..
                    xlabel('Individual'),..
                    ylabel('Fitness'),..
                    grid


      % Plot parameter k1 vs individual for this generation :
      subplot(223) ;
      ind_num = 1 : popsize ;
      axis([ 0  popsize  kg_scale(1,1)  kg_scale(1,2) ] ) ;
      plot( ind_num, new_param(:,1) ),
                    title(['Generation Number  ', int2str(gen)] ),..
                    xlabel('Individual'),..
                    ylabel('Parameter: k1'),..
                    grid

      % Plot parameter k2 vs individual for this generation :
      subplot(222) ;
      ind_num = 1 : popsize ;
      axis([ 0  popsize  kg_scale(2,1)  kg_scale(2,2) ] ) ;
      plot( ind_num, new_param(:,2) ),
                    title(['Generation Number  ', int2str(gen)] ),..
                    xlabel('Individual'),..
                    ylabel('Parameter: k2'),..
                    grid

      % Plot parameter k3 vs individual for this generation :
      subplot(224) ;
      ind_num = 1 : popsize ;
      axis([ 0  popsize  kg_scale(3,1)  kg_scale(3,2) ] ) ;
      plot( ind_num, new_param(:,3) ) ,
                    title(['Generation Number  ', int2str(gen)] ),..
                    xlabel('Individual'),..
                    ylabel('Parameter: k3'),..
                    grid


      if auto_plot == 1, print, shg, end ;

     else

      generation = [ gen, avg_fitness, max_fitness, min_fitness ]
```

```
    end ;


while  1          % operator control loop

  % Genetic Algorithm Optimization Loop

  for igen = 1 : print_gen

    gen = gen + 1 ;

    % Save Old Generation

    old_fitness = new_fitness ;

    old_chrom = new_chrom ;

    old_param = new_param ;

    new_chrom = [ ] ;

    sum_fitness = sum( old_fitness ) ;

    % Squeeze Limits Algorithm

    if squeeze_lim == 1

      for np = 1 : numparams

        last_lower = kg_scale( np, 1 ) ;
        last_upper = kg_scale( np, 2 ) ;

        data_lower = min( old_param( :, np )) ;
        data_upper = max( old_param( :, np )) ;

        [ best_fitness, imax ] = max( old_fitness ) ;
        best_param = old_param( imax, np ) ;

        % Squeeze limits toward best parameter value

        new_lower = best_param - ..
                    squeeze_factor * abs( last_lower - best_param );

        new_upper = best_param + ..
                    squeeze_factor * abs( last_upper - best_param );

        % New limits cannot exclude old data

        if new_lower > data_lower
          new_lower = data_lower ;
        end ;

        if new_upper < data_upper
          new_upper = data_upper ;
```

```
      end ;

      % New limits must maintain a buffer distance from the best
      %   parameter value ( Note - the best value can thus push
      %   the limits outward ! )


      if new_lower > ( best_param - squeeze_buffer )
        new_lower = best_param - squeeze_buffer ;
      end ;

      if new_upper < ( best_param + squeeze_buffer )
        new_upper = best_param + squeeze_buffer ;
      end ;

      % Round to integer value ( toward + / - infinity )

      new_upper = ceil( new_upper ) ;

      new_lower = floor( new_lower ) ;

      % Do not let limits exceed initial values

      if new_lower < init_scale( np, 1 )
        new_lower = init_scale( np, 1 );
      end ;

      if new_upper > init_scale( np, 2 )
        new_upper = init_scale( np, 2 );
      end ;

%     out_put = [ igen, np, best_param, new_lower, new_upper ]
%     pause

      kg_scale( np, 1 ) = new_lower ;

      kg_scale( np, 2 ) = new_upper ;

      % Recompute chromosome strings based on new scale values

      for ipo = 1 : popsize

        new_string = conv2str( old_param( ipo, np ), ..
                                  new_lower, new_upper ) ;
        if np == 1
          old_chrom( ipo, 1:16 ) = new_string ;
        elseif np == 2
          old_chrom( ipo, 17:32 ) = new_string ;
        elseif np == 3
          old_chrom( ipo, 33:48 ) = new_string ;
        end ;

      end ;  % for ipo loop

    end ;  %  for np loop
```

```
end ;  %  End of Squeeze limits

% Create New Generation

% Copy top ranked individuals exactly into next generation

if topsize > 0

  [ rank_fitness, rank_index ] = sort( old_fitness ) ;

  top_chrom = old_chrom( rank_index( popsize - topsize + 1 ..
                                                  : popsize ), : ) ;

  new_chrom = [ new_chrom ; top_chrom ] ;

end ;    % ( if topsize )

% Generate number of new random individuals in next generation

if newsize > 0

  random_chrom = fix( 1.9999999 * rand( newsize, stringlength ) );

  new_chrom = [ new_chrom ; random_chrom ] ;

end ;    % ( if newsize )

% Generate remaining number of individuals by mating parents
%  of previous generation

nloops = fix( ( popsize - topsize - newsize ) / 2 ) ;

for i = 1 : nloops

  %  - Select Parents

  [ p_1, p_2 ] = select( popsize, sum_fitness, old_fitness ) ;

  parent_1 = old_chrom( p_1, : ) ;

  parent_2 = old_chrom( p_2, : ) ;

  %  - Create 2 Children, and Perform Crossover

  child_1 = parent_1 ;

  child_2 = parent_2 ;


  %   - Perform Crossover separately on each parameter

  for ip = 1 : numparams
```

```
    if rand <= pcross
      if xover_flag == 1
        xsite = selxsite( paramlength, gen ) ;
      else
        xsite = fix( rand * ( paramlength - 1 )) + 1 ;
      end ;
      x1 = ( ip - 1 ) * paramlength + 1 ;
      x2 = x1 + xsite - 1 ;
      child_1( x1 : x2 ) = parent_2( x1 : x2 )  ;
      child_2( x1 : x2 ) = parent_1( x1 : x2 )  ;
    end ;  % ( if rand )
  end ;   % ( for ip )

  % Perform bit by bit mutation on each child

  mu_string = rand( 1, stringlength ) ;

  index = find( mu_string <= pmutation ) ;

  child_1( index ) = ~ child_1( index ) ;

  mu_string = rand( 1, stringlength ) ;

  index = find( mu_string <= pmutation ) ;

  child_2( index ) = ~ child_2( index ) ;

  new_chrom = [ new_chrom ; child_1 ; child_2 ] ;

end ;   % (  For i )


% Evaluate Fitness of New Population

[new_fitness, new_param] = evalpop7( A, B, C, D, ts, dts, tl, dtl,..
  hoe_thresh, phi_rise, act_thresh, popsize, kg_scale, new_chrom );

% Option to combine old population with new population and only
%   save best individuals from both

if combine_flag == 1

  combine_fitness = [ old_fitness ; new_fitness ] ;

  combine_chrom = [ old_chrom ; new_chrom ] ;

  combine_param = [ old_param ; new_param ] ;

  combine_size = 2 * popsize ;

  [ rank_fitness, rank_index ] = sort( combine_fitness ) ;

  new_index =  rank_index( combine_size - popsize + 1 : ..
                                              combine_size ) ;
```

```
  new_chrom = combine_chrom( [ new_index ], : ) ;

  new_param = combine_param( [ new_index ], : ) ;

  new_fitness = combine_fitness( [ new_index ] ) ;

  % Debug output :
  % rank_fitness, pause
  % rank_index, pause
  % combine_fitness, pause
  % new_fitness, pause
  % combine_param, pause
  % new_param, pause

end ;

% Evaluate Population Statistics

avg_fitness = sum( new_fitness ) / popsize ;

[ max_fitness, imax ] = max( new_fitness ) ;

max_param = new_param( imax, : ) ;

max_chrom = new_chrom( imax, : ) ;

[ min_fitness, imin ] = min( new_fitness ) ;

min_param = new_param( imin, : ) ;

min_chrom = new_chrom( imin, : ) ;


% Save generational data :

gen_avg_fitness = [ gen_avg_fitness ; avg_fitness ] ;

gen_max_fitness = [ gen_max_fitness ; max_fitness ] ;

gen_max_param = [ gen_max_param ; max_param ] ;

gen_max_chrom = [ gen_max_chrom ; max_chrom ] ;

gen_min_fitness = [ gen_min_fitness ; min_fitness ] ;

gen_min_param = [ gen_min_param ; min_param ] ;

gen_min_chrom = [ gen_min_chrom ; min_chrom ] ;

gen_kg_lower = [ gen_kg_lower ; kg_scale( :, 1 ).' ] ;

gen_kg_upper = [ gen_kg_upper ; kg_scale( :, 2 ).' ] ;
```

```
% Display or plot values to screen

if cont_plot == 1

  clg ;

  % Plot of fitness vs individual for this generation :
  subplot(221) ;
  ind_num = 1 : popsize ;
  axis ;   % reset scaling to automatic
  plot( ind_num, new_fitness ),
                title(['Generation Number  ', int2str(gen)] ),..
                xlabel('Individual'),..
                ylabel('Fitness'),..
                grid


  % Plot parameter k1 vs individual for this generation :
  subplot(223) ;
  ind_num = 1 : popsize ;
  axis([ 0, popsize, kg_scale(1,1), kg_scale(1,2) ] ) ;
  plot( ind_num, new_param(:,1) ),
                title(['Generation Number  ', int2str(gen)] ),..
                xlabel('Individual'),..
                ylabel('Parameter: k1'),..
                grid

  % Plot parameter k2 vs individual for this generation :
  subplot(222) ;
  ind_num = 1 : popsize ;
  axis([ 0, popsize, kg_scale(2,1), kg_scale(2,2) ] ) ;
  plot( ind_num, new_param(:,2) ),
                title(['Generation Number  ', int2str(gen)] ),..
                xlabel('Individual'),..
                ylabel('Parameter: k2'),..
                grid

  % Plot parameter k3 vs individual for this generation :
  subplot(224) ;
  ind_num = 1 : popsize ;
  axis([ 0, popsize, kg_scale(3,1), kg_scale(3,2) ] ) ;
  plot( ind_num, new_param(:,3) ),
                title(['Generation Number  ', int2str(gen)] ),..
                xlabel('Individual'),..
                ylabel('Parameter: k3'),..
                grid

 else

   generation = [ gen, avg_fitness, max_fitness, min_fitness ]

 end ;

end ;   %  ( for igen loop )
```

```
    if auto_plot == 1, print, shg, end ;  % output to printer

%     char = input( ' To continue with more generations enter : y ', 's'
)
%     if strcmp( char, 'y' ) == 1
%        shg ;
%     else
%        break ;
%     end ;

   if gen >= maxgen, break, end;  % break operator control loop

   end ;  % ( while - operator control loop )


   subplot    % return to default, i.e. full screen plots


  %  Plot of Maximum Fitness Function :

      gen_num = 1 : gen ;
      axis ;   % reset scaling to automatic
      plot( gen_num, gen_max_fitness ),
                     title('Maximum Fitness vs. Generation'),..
                     xlabel('Generation'),..
                     ylabel('Maximum Fitness'),..
                     grid


  %  Always output Maximum Fitness Function :

        print ;


  %  Plot of best parameter vs generation number :

      gen_num = 1 : gen ;

      genp1 = gen + 1 ;

      axis([ 0, genp1, -100, 20 ] ) ;
      plot( gen_num, gen_max_param( :,1),'-',..
            gen_num, gen_max_param( :,2),'--',..
            gen_num, gen_max_param( :,3),'-.'),..
                     title('Best Parameters vs. Generation'),..
                     xlabel('Generation'),..
                     ylabel('Best Parameters: k1, k2, k3'),..
                     text( gen + .25 , gen_max_param(gen,1), 'k1'),..
                     text( gen + .25 , gen_max_param(gen,2), 'k2'),..
                     text( gen + .25 , gen_max_param(gen,3), 'k3'),..
                     grid

  % Always output best parameters function :
```

```
   print ;

% Output final best parameters and fitness data to screen :

gen
gen_max_fitness( gen )
gen_max_param( gen, : )
pause


%  Plot of Average Fitness Function :

     gen_num = 1 : gen ;
     axis ;   % reset scaling to automatic
     plot( gen_num, gen_avg_fitness ),
                    title('Genetic Algorithm - Average Fitness'),..
                    xlabel('Generation'),..
                    ylabel('Average Fitness'),..
                    grid,pause

% Prompt for hard copy plot:
     char = input(' To make hardcopy of plot enter: y','s')  ;
     if strcmp( char, 'y' ) == 1
       print
     end ;


%     %  Plot of kg scale limits vs generation :
%          gen_num = 1 : gen ;
%    plot( gen_num, gen_kg_lower(:,1), '-', gen_num, gen_kg_upper(:,1), '-',..
%          gen_num, gen_kg_lower(:,2),'--', gen_num, gen_kg_upper(:,2),'--',..
%          gen_num, gen_kg_lower(:,3),'-.', gen_num, gen_kg_upper(:,3),'-.'),
%                          title('- Gain Scale Limits -'),..
%                          xlabel('Generation'),..
%                          ylabel('Limits'),..
%                          grid, pause

%     % Prompt for hard copy plot:
%          char = input(' To make hardcopy of plot enter: y','s')  ;
%          if strcmp( char, 'y' ) == 1
%            print
%          end ;

% Set the gain to the maximum value of the last generation :

    kgain = gen_max_param( gen, : )

% Use program AP5 to evaluate these gains !
  pause

 clc
 % APGEN3 - program completed !
```

A.2 AP5

This MATLAB program builds the airframe and linear feedback controller state space model and provides several functions for off-line analysis and plots. It uses the Control System Toolbox LQR2 fuction to determine the optimum steady state Linear Quadratic Regulator design by solving the associated continuous-time Riccati equation. It is also used to analyze other designs by manually overriding the LQR gains.

This program then generates a step response of the resulting closed loop control system and provides analysis of various cost and fitness measures. Next, it generates step response plots of the state and control variables. Finally, it provides frequency domain analysis and plots of the open loop control system.

A listing of this program follows.

```
echo on
clc
%     AP5  - Autopilot Design and Analysis Program # 5
%          - airframe model only
%          - with linear feedback controller
%          - for LQR design
%             by    R. Hull,    10/5/92
%
%          - corrected for N_delta sign error in B matrix
%               1/28/93
%          - modified to use the step2 step response function
%               2/1/93
%          - modified to use the step3 response function 2/4/93
%          - back to step2 function 2/8/93
%          - use step2s to compute dual dt step response 3/1/93
%
%  This program builds the linearized missile airframe model
%    with unity accelerometer and gyro feedback models.
%
%
%    Special Constants :

     HztR = 2*pi;          % Converts Hz to Radians per Second

%    Flight Condition Constants :
```

```
    M_alpha = 64.11;
%   M_delta = -20.00        % Reduced fin effectiveness
    M_delta = -62.34;
    M_thetadot = 0;
    N_alpha = .1803;
    N_delta = .0738;
    Tau_act = .02 ;         % First order actuator time constant
    Velocity = 892.0;       % meters/sec
    Flight_Time = 30.0;     % seconds

% Units Conversions

    format compact ;

% First - create plant model without computation delay and generate
%         LQR controller gains

% Define Airframe + First Order Actuator State Space Realization :

    A = [ - N_alpha  1.0  - N_delta
          M_alpha    0      M_delta
            0        0    - ( 1 / Tau_act ) ] ;

    B = [    0
             0
           ( 1 / Tau_act ) ]

    C = [ ( N_alpha * Velocity )  0   ( N_delta * Velocity ) ] ;


    D = [ 0 ] ;


% Determine LQR design for basic plant with first order actuator :

    qmatrix = [  10  0  0
                  0 .1  0
                  0  0 .01  ];

    rmatrix = .01;

  %  kgain = lqr2(A,B,qmatrix,rmatrix);

    % LQR design gains :
    %  kgain
    %  pause


%    kgain = [ -34.1723  -3.6403  2.3434 ] ;  % LQR gains
%  Genetic Algorithm Gains:
%    kgain = [ -48.9113  -6.2638  1.6400 ] ;  % J Controller
%    kgain = [ -66.0487  -14.9935  9.1844 ] ;  % J,Pd Controller
%    kgain = [ -24.7425  -7.0481  1.6289 ] ;  % J,Pd,Psettle Controller
```

```
kgain = [ -22.3987  -4.5464  1.6514 ] ; % J,Pd,Psettle,Prise Controller

   % Now - formulate open loop control system for
   %       linear state feedback control
   %       and determine gain and phase margins of open loop system

        Aopen = A

        Bopen = B

        Copen = - kgain

        Dopen =  [ 0 ]


   % Now - formulate closed loop regulator with
   %       linear state feedback control
   %       to determine step response of closed loop system

        hoec = 1.0              % hang off error compensation

        Ac = A - B * kgain

        Bc = hoec * B * kgain

        Cc = C - D * kgain

        Dc = hoec * D * kgain

   % Set Cc & Dc matrices to force output equal to control

        Cc = - kgain

        Dc = hoec * kgain

   % Define hard limits for state responses :
   %    ( used in step3 function )

        Slim = [ - Inf   Inf
                 - Inf   Inf
                 - 100.0  100.0 ] ;

  % Gains used :
    kgain

    pause      %  Strike any key for step response ...


    % Now compute the step response :

        dtl = .01 ;

        tl_final = 5.0 ;
```

```
      dts = .001 ;

      ts_final = 0.2 ;

      ts = 0 : dts : ts_final ;

      tl = ts_final : dtl : tl_final ;

   % Now compute the step response :

      Rc = [      1
                  0
                  0       ] ;      % Reference Input Commands

      [ys,yl,xs,xl] = step2s(Ac,Bc,Cc,Dc,Rc,dts,ts,dtl,tl);


    % Evaluate the cost function separately for small and large dt
    %   responses and sum to get total :

        r = [ 1   0   0 ] ;    % indicates desired step response

        j1 = jcost( xs, r, ys, qmatrix, rmatrix, dts );

        j2 = jcost( xl, r, yl, qmatrix, rmatrix, dtl );

        j = j1 + j2 ;

    % Evaluate the rise time

        lens = length( ts ) ;

        t_rise = risetime( xs(:,1), ts, .8 ) ;

        if t_rise >= ts( lens )
          t_rise = risetime( xl(:,1), tl, .8 ) ;
        end ;

    % Determine infinity norm of actuator response

        act_norm = infnorm( xs(:,3) ) ;

    % Determine the Hang Off Error in state 1 ( final value error ) :

        [ xrows, xcols ] = size( xl ) ;

        hang_off_error = xl( xrows, 1 ) - r( 1 ) ;

% Gains used :
  kgain

% Cost function :
  j

% Rise Time :
```

```
t_rise

% Settling error :
hang_off_error

% Actuator peak response
act_norm


% Strike Enter to Actuator Step Response ...
pause ;

     clg
     axis([0,ts_final,-10,10])
     plot(ts,xs(:,3)), title('Actuator Step Response'),..
                       xlabel('Time - sec'),..
                       ylabel('Actuator Angle - deg'),..
                       grid,pause

% Prompt for hard copy plot:

     char = input(' To make hardcopy of plot enter: y','s')

     if strcmp( char, 'y' ) == 1
       print
     end


%  Steady State Step Angle of Attack Response :
     clg
     axis([0,tl_final,0,2])
     plot(ts,xs(:,1),tl,xl(:,1)),..
               title('Angle of Attack Step Response'),..
               xlabel('Time - sec'),..
               ylabel('Angle of Attack - deg'),..
               grid,pause

% Prompt for hard copy plot:

     char = input(' To make hardcopy of plot enter: y','s')

     if strcmp( char, 'y' ) == 1
       print
     end

% Strike any key - for Gain and Phase Margins ...
pause

echo off
     w = logspace(-1,7,160);
     [mag,phase] = bode( Aopen,Bopen,Copen,Dopen,1,w);
     [LGm,HGm,Pm,WLg,WHg,Wcp] = margin2( mag, phase, w )

% Note - MARGIN2 is just a modified form of MARGIN to accept
%         mag in dB.
```

```
fprintf( ..
   '  Freq  =  %8.2f rad/sec   Low Freq Gain Margin = %8.2f dB\n',..
        WLg, LGm )

fprintf( ..
   '  Freq  =  %8.2f rad/sec  High Freq Gain Margin = %8.2f dB\n',..
        WHg, HGm )

fprintf( ..
   '  Freq  =  %8.2f rad/sec  Phase Margin = %8.2f deg\n', Wcp, Pm )

echo on


% Strike any key - for Bode Plot ...
pause

clg
subplot(211)
axis([-1,7,-100,100])
semilogx(w,mag), title('Open Loop CAS delta frequency response'),..
                 xlabel('Radians / sec'),..
                 ylabel('Gain - dB'),..
                 grid,..
subplot(212)
axis([-1,7,-200,200])
semilogx(w,phase), title('Phase response')
                   xlabel('Radians / sec'),..
                   ylabel('Phase - Deg'),..
                   grid,..
pause

% Prompt for hard copy plot:

     char = input(' To make hardcopy of plot enter: y','s')

     if strcmp( char, 'y' ) == 1
       print
     end


subplot    % return to default, i.e. full screen plots

% Strike any key - for Nyquist plot ...
pause

     w = logspace(-1,3,80);
     [re,im] = nyquist( Aopen,Bopen,Copen,Dopen,1,w);

     clg
     axis
     plot(re,im), title('Open Loop CAS delta Nyquist Plot'),..
                  xlabel('real axis'),..
                  ylabel('imaginary axis'),..
```

```
                    grid,..
pause

% Prompt for hard copy plot:

     char = input(' To make hardcopy of plot enter: y','s')

     if strcmp( char, 'y' ) == 1
       print
     end


% Strike any key -  for Zoomed Nyquist plot :
pause

     clg
     axis([-10,0,-5,5])
     plot(re,im), title('Open Loop CAS delta Nyquist Plot'),..
                    xlabel('real axis'),..
                    ylabel('imaginary axis'),..
                    grid,..
pause

% Prompt for hard copy plot:

     char = input(' To make hardcopy of plot enter: y','s')

     if strcmp( char, 'y' ) == 1
       print
     end

 clc

 % AP5 - program completed !
```

A.3 APNMOPT

This MATLAB program determines the true optimum fitness function for a given controller specification, using the MATLAB function FMINS, which implements a Nelder-Meade numerical search algorithm.

A listing of this program follows.

```
echo on,  clc
%     APNMOPT.M  - Autopilot Nelder-Meade Optimizaton Program
%                - with linear feedback controller
%                - numerical optimization of autopilot controller
%                    fitness function using Nelder-Meade simplex
%                    algorithm in Matlab function FMINS.
%
%                by    R. Hull,    4/6/93

     kg = [ -48.9113 -6.2638 1.64 ]

     fit_test = - apfitfun( kg' )

     kg_opt = fmins( 'apfitfun', kg' )

     fitness = - apfitfun( kg_opt )

     pause   % APNMOPT - program completed !
```

A.4 APOPLOT

This MATLAB program plots the optimum fitness functions parametrically vs one gain, while holding the other two constant. It overlays the genetically optimized fitness function with the true optimum fitness function for comparison.

A listing of this program follows.

```
echo on
clc
%      APOPLOT.M  - Program to plot optimum fitness function
%
%                 by    R. Hull,    4/6/93
%

  format compact ;

  ga_opt = [ -22.3987 -4.5464 1.6514 ]   % Input GA Optimum Gains

  ga_fit = 1.2367                        % Input GA max fitness

  true_opt = [ -22.2218 -4.4455 1.4819 ]   % Input True Optimum Gains

  true_fit = 1.3316                       % Input True max fitness

  y_scale = 1.8 ;                         % Max y axis scale for plot

%  Note - change controller spec in APFITFUN to match gains above !
   pause

   k1_low = -30 ;
   k1_high = -10 ;
   k1_delta = 1 ;

   k2_low = -11 ;
   k2_high = -1 ;
   k2_delta = .5 ;

   k3_low = 0.1 ;
   k3_high = 4.1 ;
   k3_delta = .2 ;

% Option to skip k1 loop

 char = input(' To skip k1 plot enter y: y','s')

 if strcmp( char, 'y' ) == 0
```

```
k2_ga = ga_opt( 2 );        % for fixed value of k2 gain

k3_ga = ga_opt( 3 );        % for fixed value of k3 gain

k2_true = true_opt( 2 );        % for fixed value of k2 gain

k3_true = true_opt( 3 );        % for fixed value of k3 gain

%  Create k1 history to contain values at ga optimum and true optimum

k1_temp1 = k1_low : k1_delta : k1_high ;

k1_insert = ga_opt(1) ;

k1_temp2 = [ k1_temp1( k1_temp1 < k1_insert) k1_insert ..
             k1_temp1( k1_temp1 > k1_insert) ] ;

k1_insert = true_opt(1) ;

k1_hist = [ k1_temp2( k1_temp2 < k1_insert) k1_insert ..
             k1_temp2( k1_temp2 > k1_insert) ] ;

% Initialize for ki loop

   max_fitness = 0.0 ;

   ga_fit_hist = [ ] ;

   true_fit_hist = [ ] ;

   k1_len = length( k1_hist ) ;

   for ki = 1 : k1_len

     k1 = k1_hist( ki ) ;

     kg_ga = [ k1 k2_ga k3_ga ] ;

     kg_true = [ k1 k2_true k3_true ] ;

     ga_fitness = - apfitfun( kg_ga' ) ;

     true_fitness = - apfitfun( kg_true' ) ;

     %  Save data for plots

     ga_fit_hist = [ ga_fit_hist ; ga_fitness ] ;

     true_fit_hist = [ true_fit_hist ; true_fitness ] ;

     data_out = [ k1 ga_fitness true_fitness ]  % output to screen
```

```
    end     % for ki loop

    k1

%  Strike any key for plot of Fitness Function :
    pause

    clg

    axis([ k1_low, k1_high, 0, y_scale ] ) ;
    plot( k1_hist, ga_fit_hist, ga_opt(1), ga_fit, 'x', ..
          k1_hist, true_fit_hist, true_opt(1), true_fit, '*'  ),..
        title('Fitness Function vs k1'),..
        xlabel('k1 gain'),..
        ylabel('Fitness'),..
        text( ga_opt(1), ga_fit + .1, 'GA Optimum' ),..
        text( true_opt(1), true_fit + .15, 'True Optimum' ),..
        grid,pause

% Prompt for hard copy plot:

    char = input(' To make hardcopy of plot enter: y','s')

    if strcmp( char, 'y' ) == 1
      print
    end

 end    % end if  - option to skip k1 loop


% Option to skip k2 loop

 char = input(' To skip k2 plot enter y: y','s')

 if strcmp( char, 'y' ) == 0

   k1_ga = ga_opt( 1 );      % for fixed value of k1 gain

   k3_ga = ga_opt( 3 );      % for fixed value of k3 gain

   k1_true = true_opt( 1 );      % for fixed value of k1 gain

   k3_true = true_opt( 3 );      % for fixed value of k3 gain


%  Create k2 history to contain values at ga optimum and true optimum

   k2_temp1 = k2_low : k2_delta : k2_high ;

   k2_insert = ga_opt(2) ;

   k2_temp2 = [ k2_temp1( k2_temp1 < k2_insert) k2_insert ..
                k2_temp1( k2_temp1 > k2_insert) ] ;

   k2_insert = true_opt(2) ;
```

```
   k2_hist = [ k2_temp2( k2_temp2 < k2_insert) k2_insert ..
               k2_temp2( k2_temp2 > k2_insert) ] ;

% Initialize for ki loop

     max_fitness = 0.0 ;

     ga_fit_hist = [ ] ;

     true_fit_hist = [ ] ;

     k2_len = length( k2_hist ) ;


     for ki = 1 : k2_len

       k2 = k2_hist( ki ) ;

       kg_ga = [ k1_ga k2 k3_ga ] ;

       kg_true = [ k1_true k2 k3_true ] ;

       ga_fitness = - apfitfun( kg_ga' ) ;

       true_fitness = - apfitfun( kg_true' ) ;

       %  Save data for plots

       ga_fit_hist = [ ga_fit_hist ; ga_fitness ] ;

       true_fit_hist = [ true_fit_hist ; true_fitness ] ;

       data_out = [ k2 ga_fitness true_fitness ]  % output to screen

     end     % for ki loop

     k2

%  Strike any key for plot of Fitness Function :
     pause

     clg

     axis([ k2_low, k2_high, 0, y_scale ] ) ;
     plot( k2_hist, ga_fit_hist, ga_opt(2), ga_fit, 'x', ..
           k2_hist, true_fit_hist, true_opt(2), true_fit, '*'  ),..
         title('Fitness Function vs k2'),..
         xlabel('k2 gain'),..
         ylabel('Fitness'),..
         text( ga_opt(2), ga_fit + .1, 'GA Optimum' ),..
         text( true_opt(2), true_fit + .15, 'True Optimum' ),..
         grid,pause
```

```
% Prompt for hard copy plot:

     char = input(' To make hardcopy of plot enter: y','s')

     if strcmp( char, 'y' ) == 1
       print
     end

 end    % end if  - option to skip k2 loop


% Option to skip k3 loop

 char = input(' To skip k3 plot enter y: y','s')

 if strcmp( char, 'y' ) == 0


   k1_ga = ga_opt( 1 );       % for fixed value of k1 gain

   k2_ga = ga_opt( 2 );       % for fixed value of k2 gain

   k1_true = true_opt( 1 );       % for fixed value of k1 gain

   k2_true = true_opt( 2 );       % for fixed value of k2 gain


%  Create k3 history to contain values at ga optimum and true optimum

   k3_temp1 = k3_low : k3_delta : k3_high ;

   k3_insert = ga_opt(3) ;

   k3_temp2 = [ k3_temp1( k3_temp1 < k3_insert) k3_insert ..
                k3_temp1( k3_temp1 > k3_insert) ] ;

   k3_insert = true_opt(3) ;

   k3_hist = [ k3_temp2( k3_temp2 < k3_insert) k3_insert ..
                k3_temp2( k3_temp2 > k3_insert) ] ;


% Initialize for ki loop

     max_fitness = 0.0 ;

     ga_fit_hist = [ ] ;

     true_fit_hist = [ ] ;

     k3_len = length( k3_hist ) ;


     for ki = 1 : k3_len
```

```
      k3 = k3_hist( ki ) ;

      kg_ga = [ k1_ga k2_ga k3 ] ;

      kg_true = [ k1_true k2_true k3 ] ;

      ga_fitness = - apfitfun( kg_ga' ) ;

      true_fitness = - apfitfun( kg_true' ) ;

      %  Save data for plots

      ga_fit_hist = [ ga_fit_hist ; ga_fitness ] ;

      true_fit_hist = [ true_fit_hist ; true_fitness ] ;

      data_out = [ k3 ga_fitness true_fitness ]  % output to screen

    end     % for ki loop

    k3

%  Strike any key for plot of Fitness Function :
    pause

    clg

    axis([ k3_low, k3_high, 0, y_scale ] ) ;
    plot( k3_hist, ga_fit_hist, ga_opt(3), ga_fit, 'x', ..
          k3_hist, true_fit_hist, true_opt(3), true_fit, '*'  ),..
        title('Fitness Function vs k3'),..
        xlabel('k3 gain'),..
        ylabel('Fitness'),..
        text( ga_opt(3), ga_fit + .1, 'GA Optimum' ),..
        text( true_opt(3), true_fit + .15, 'True Optimum' ),..
        grid,pause

% Prompt for hard copy plot:

    char = input(' To make hardcopy of plot enter: y','s')

    if strcmp( char, 'y' ) == 1
      print
    end

 end    % end if  - option to skip k3 loop

 % APOPLOT - program completed !
```

A.5 GATEST

This MATLAB program implements a simple genetic algorithm to demonstrate its effectiveness on a single parameter example with a non-trivial solution.

A listing of this program follows.

```
echo on
clc
%          GATEST  - Genetic Algorithm Test Program
%               - optimize special function of one parameter
%
%                 by    R. Hull,    2/19/93
%
%
%    Special Constants :

     HztR = 2*pi;          % Converts Hz to Radians per Second


%   Global Variables :

     global cont_plot ;

     cont_plot = 1 ;     %   Continuous plot flag ( on monitor )

     auto_plot = 1 ;     %   Automatic plot flag ( to printer )

     combine_flag = 0 ; % Option to combine old and new populations
                        %    and save the best individuals of both.

%    Genetic Algorithm Structures

     maxgen = 4 ;       %  number of generations per loop

     popsize = 20 ;     % number of individules in population
                        %  must be even !

     topsize = 2 ;      % number ( even ) of top ranked individuals to
                        %    propagate exactly in the next generation

     if combine_flag == 1  % Make sure topsize is 0 if combine option
       topsize = 0 ;       %   is used.
     end ;

     newsize = 0 ;      % number ( even ) of new random individuals to
                        %    include in each new generation

     numparams = 1 ;  % number of parameters to optimize
```

```
    paramlength = 16 ;  % number of bits per parameter
    stringlength = numparams * paramlength ;    %  string length
                                                %  ( bits )
    t_final = 5.0 ;          % fitness function final time - sec.
    pcross = .8 ;            % probability of crossover
    pmutation = .025         % probability of mutation
    kg_scale = [ 0  5  ]     % gain parameter scales

% Units Conversions

% Initialize time values
    dt = .01 ;
    t = 0:dt:t_final ;
% Set screen display format to suppress excessive line feeds
    format compact ;
% Initialize New Population to Random Strings
    new_chrom = fix( 1.9999999 * rand( popsize, stringlength ) ) ;

 %  Clear generational statistics
 gen_avg_fitness = [ ] ;
 gen_max_fitness = [ ] ;
 gen_max_param = [ ] ;
 gen_max_chrom = [ ] ;
 gen_min_fitness = [ ] ;
 gen_min_param = [ ] ;
 gen_min_chrom = [ ] ;
 gen_kg_lower = [ ] ;
 gen_kg_upper = [ ] ;
 gen = 1 ;     % generation number
```

```
% Evaluate Fitness of Initial Population

[new_fitness, new_param] = evalpopt( popsize, kg_scale, new_chrom );

% Evaluate Initial Population Statistics

avg_fitness = sum( new_fitness ) / popsize ;

[ max_fitness, imax ] = max( new_fitness ) ;

max_param = new_param( imax ) ;

max_chrom = new_chrom( imax ) ;

[ min_fitness, imin ] = min( new_fitness ) ;

min_param = new_param( imin ) ;

min_chrom = new_chrom( imin ) ;

% Save initial generation data :

gen_avg_fitness = [ gen_avg_fitness ; avg_fitness ] ;

gen_max_fitness = [ gen_max_fitness ; max_fitness ] ;

gen_max_param = [ gen_max_param ; max_param ] ;

gen_max_chrom = [ gen_max_chrom ; max_chrom ] ;

gen_min_fitness = [ gen_min_fitness ; min_fitness ] ;

gen_min_param = [ gen_min_param ; min_param ] ;

gen_min_chrom = [ gen_min_chrom ; min_chrom ] ;

%  Plot of Fitness Function vs x :

    fit_t = [ ] ;

    t = 0:.01:5.0 ;

    for i = 1 : length(t)
      fit =  - ( t(i) - 2.5 ) ^ 2 + 10.0 + 5 * sin( 4 * pi * t(i) );
      fit_t = [ fit_t ; fit ] ;
    end;

    plot( t, fit_t ),
                    title('Fitness Function'),..
                    xlabel('x'),..
                    ylabel('Fitness'),..
                    grid,pause

% Prompt for hard copy plot:
```

```
char = input(' To make hardcopy of plot enter: y','s')  ;

if strcmp( char, 'y' ) == 1
  print
end ;


% Display or plot values to screen

pause

if cont_plot == 1

  clg ;

  % Plot of fitness vs individual for this generation :
  subplot(221) ;
  ind_num = 1 : popsize ;
  plot( ind_num, new_fitness ),
          title(['Generation Number  ', int2str(gen)] ),...
          xlabel('Individual'),...
          ylabel('Fitness'),...
          grid


  % Plot of Maximum fitness vs generation :
  subplot(222) ;
  gen_num = 1 : gen ;
  plot( gen_num, gen_max_fitness ),
          title('- Maximum Fitness -'),...
          xlabel('Generation'),...
          ylabel('Maximum Fitness'),...
          grid


  % Plot parameter vs individual for this generation :
  subplot(223) ;
  ind_num = 1 : popsize ;
  plot( ind_num, new_param ),
          title(['Generation Number  ', int2str(gen)] ),...
          xlabel('Individual'),...
          ylabel('Parameter: x'),...
          grid


  %  Plot of Average Fitness vs generation :
  subplot(224) ;
  gen_num = 1 : gen ;
  plot( gen_num, gen_avg_fitness ),
           title('- Average Fitness -'),...
           xlabel('Generation'),...
           ylabel('Average Fitness'),...
           grid,pause
```

```
%    if auto_plot == 1, print, shg, end ;

   % Prompt for hard copy plot:
   char = input(' To make hardcopy of plot enter: y','s')  ;
   if strcmp( char, 'y' ) == 1 , print, end ;

  else

    generation = [ gen, avg_fitness, max_fitness, min_fitness ]

  end ;

while  1          % operator control loop

 % Genetic Algorithm Optimization Loop

 for igen = 1 : maxgen

   gen = gen + 1 ;

   % Save Old Generation

   old_fitness = new_fitness ;

   old_chrom = new_chrom ;

   old_param = new_param ;

   new_chrom = [ ] ;

   sum_fitness = sum( old_fitness ) ;


   % Create New Generation

   % Copy top ranked individuals exactly into next generation

   if topsize > 0

     [ rank_fitness, rank_index ] = sort( old_fitness ) ;

     top_chrom = old_chrom( rank_index( popsize - topsize + 1 ..
                                              : popsize ), : ) ;

     new_chrom = [ new_chrom ; top_chrom ] ;

   end ;     % ( if topsize )


   % Generate number of new random individuals in next generation

   if newsize > 0

     random_chrom = fix( 1.9999999*rand(newsize, stringlength));
```

```
  new_chrom = [ new_chrom ; random_chrom ] ;
end ;    % ( if newsize )

% Generate remaining number of individuals by mating parents
%  of previous generation
nloops = fix( ( popsize - topsize - newsize ) / 2 ) ;
for i = 1 : nloops
  %  - Select Parents
  [ p_1, p_2 ] = select( popsize, sum_fitness, old_fitness ) ;
  parent_1 = old_chrom( p_1, : ) ;
  parent_2 = old_chrom( p_2, : ) ;
  %  - Create 2 Children, and Perform Crossover
  child_1 = parent_1 ;
  child_2 = parent_2 ;
  %   - Perform Crossover separately on each parameter
  for ip = 1 : numparams
    if rand <= pcross
      xsite = fix( rand * ( paramlength - 1 )) + 1 ;
      x1 = ( ip - 1 ) * paramlength + 1 ;
      x2 = x1 + xsite - 1 ;
      child_1( x1 : x2 ) = parent_2( x1 : x2 )  ;
      child_2( x1 : x2 ) = parent_1( x1 : x2 )  ;
    end ;  % ( if rand )
  end ;   % ( for ip )
  % Perform bit by bit mutation on each child
  mu_string = rand( 1, stringlength ) ;
  index = find( mu_string <= pmutation ) ;
  child_1( index ) = ~ child_1( index ) ;
  mu_string = rand( 1, stringlength ) ;
  index = find( mu_string <= pmutation ) ;
  child_2( index ) = ~ child_2( index ) ;
  new_chrom = [ new_chrom ; child_1 ; child_2 ] ;
end ;   % (  For i )
```

```
% Evaluate Fitness of New Population

[ new_fitness, new_param ] = evalpopt( popsize, kg_scale, ..
                                                    new_chrom ) ;

% Option to combine old population with new population and only
%   save best individuals from both

if combine_flag == 1

  combine_fitness = [ old_fitness ; new_fitness ] ;

  combine_chrom = [ old_chrom ; new_chrom ] ;

  combine_param = [ old_param ; new_param ] ;

  combine_size = 2 * popsize ;

  [ rank_fitness, rank_index ] = sort( combine_fitness ) ;

  new_index =  rank_index( combine_size - popsize + 1 : ..
                                                  combine_size ) ;

  new_chrom = combine_chrom( [ new_index ], : ) ;

  new_param = combine_param( [ new_index ], : ) ;

  new_fitness = combine_fitness( [ new_index ] ) ;

end ;

% Evaluate Population Statistics

avg_fitness = sum( new_fitness ) / popsize ;

[ max_fitness, imax ] = max( new_fitness ) ;

max_param = new_param( imax ) ;

max_chrom = new_chrom( imax ) ;

[ min_fitness, imin ] = min( new_fitness ) ;

min_param = new_param( imin ) ;

min_chrom = new_chrom( imin ) ;

% Save generational data :

gen_avg_fitness = [ gen_avg_fitness ; avg_fitness ] ;
```

```
gen_max_fitness = [ gen_max_fitness ; max_fitness ] ;

gen_max_param = [ gen_max_param ; max_param ] ;

gen_max_chrom = [ gen_max_chrom ; max_chrom ] ;

gen_min_fitness = [ gen_min_fitness ; min_fitness ] ;

gen_min_param = [ gen_min_param ; min_param ] ;

gen_min_chrom = [ gen_min_chrom ; min_chrom ] ;

gen_kg_lower = [ gen_kg_lower ; kg_scale( 1 ) ] ;

gen_kg_upper = [ gen_kg_upper ; kg_scale( 2 ) ] ;


% Display or plot values to screen

if cont_plot == 1

  clg ;

  % Plot of fitness vs individual for this generation :
  subplot(221) ;
  ind_num = 1 : popsize ;
  plot( ind_num, new_fitness ),
          title(['Generation Number  ', int2str(gen)] ),...
          xlabel('Individual'),...
          ylabel('Fitness'),...
          grid


  % Plot of Maximum fitness vs generation :
  subplot(222) ;
  gen_num = 1 : gen ;
  plot( gen_num, gen_max_fitness ),
          title('- Maximum Fitness -'),...
          xlabel('Generation'),...
          ylabel('Maximum Fitness'),...
          grid


  % Plot parameter vs individual for this generation :
  subplot(223) ;
  ind_num = 1 : popsize ;
  plot( ind_num, new_param ),
          title(['Generation Number  ', int2str(gen)] ),...
          xlabel('Individual'),...
          ylabel('Parameter: x'),...
          grid


  %  Plot of Average Fitness vs generation :
  subplot(224) ;
```

```
    gen_num = 1 : gen ;
    plot( gen_num, gen_avg_fitness ),
             title('- Average Fitness -'),..
             xlabel('Generation'),..
             ylabel('Average Fitness'),..
             grid

    pause ;

%    if auto_plot == 1, print, shg, end ;

   % Prompt for hard copy plot:
   char = input(' To make hardcopy of plot enter: y','s')  ;
   if strcmp( char, 'y' ) == 1 , print, end ;


   else

    generation = [ gen, avg_fitness, max_fitness, min_fitness ]

   end ;

 end ;   %  ( for igen loop )

 gen
 gen_max_fitness( gen )
 gen_max_param( gen, : )

 char = input('To continue with more generations enter : y ', 's')

 if strcmp( char, 'y' ) == 1
    shg ;
 else
    % Prompt for hard copy plot:
    char = input(' To make hardcopy of plot enter: y','s')  ;
    if strcmp( char, 'y' ) == 1 , print, end ;
    break ;
 end ;

end ;  % ( while - operator control loop )

% Output final best parameters and fitness data to screen :

gen
gen_max_fitness( gen )
gen_max_param( gen, : )
pause

subplot    % return to default, i.e. full screen plots

%  Plot of Average Fitness Function :

    gen_num = 1 : gen ;

    plot( gen_num, gen_avg_fitness ),
```

```
            title('Genetic Algorithm - Average Fitness'),..
            xlabel('Generation'),..
            ylabel('Average Fitness'),..
            grid,pause

% Prompt for hard copy plot:

    char = input(' To make hardcopy of plot enter: y','s')  ;

    if strcmp( char, 'y' ) == 1
      print
    end ;


%  Plot of Maximum Fitness Function :

    gen_num = 1 : gen ;

    plot( gen_num, gen_max_fitness ),
            title('Genetic Algorithm - Maximum Fitness'),..
            xlabel('Generation'),..
            ylabel('Maximum Fitness'),..
            grid,pause

% Prompt for hard copy plot:

    char = input(' To make hardcopy of plot enter: y','s') ;

    if strcmp( char, 'y' ) == 1
      print
    end ;


%  Plot of best parameter vs generation number :

    gen_num = 1 : gen ;

    plot( gen_num, gen_max_param ),
            title('Genetic Algorithm - Best Parameter'),..
            xlabel('Generation'),..
            ylabel('Best Parameter: x'),..
            grid,pause

% Prompt for hard copy plot:

    char = input(' To make hardcopy of plot enter: y','s')  ;

    if strcmp( char, 'y' ) == 1
      print
    end ;

clc

% GATEST - program completed !
```

APPENDIX B

Special MATLAB Supporting Functions

Several special supporting functions were written in MATLAB and are required by one or more of the MATLAB programs described in Appendix A. This appendix briefly describes these supporting functions and provides a listing of their MATLAB source files. In addition these functions and the main programs require numerous standard MATLAB functions, as well as supporting functions from the MATLAB Control System Toolbox [13,10].

B.1 APFITFUN

This MATLAB function evaluates the fitness function for a given set of gain parameters and design specification. It first calculates the system step response, using STEP2S, and then determines the linear quadratic performance index using JCOST. It then evaluates the cost function penalties imposed by peak actuator response, settling error, and rise time constraints. The final cost function is limited and converted to a fitness function appropriate for the genetic algorithm.

A program listing of this function follows.

```
%   FUNCTION APFITFUN  -  Autopilot Fitness Function
%                - Note: this function duplicates much
%                    of the code in EVALPOP7
%                - Computes the Fitness Function based
%                    on three input gain parameters ( kg )
%                - Inverts final fitness value for use
%                    by MATLAB fmins optimization function.
%
%              by : R. Hull  4/6/93
%
function  fitness = apfitfun( kg_vector ) ;

     kg = kg_vector' ;      % convert from column vector required by
                            %  function fmins

%    Flight Condition Constants :

     M_alpha = 64.11;
     M_delta = -62.34;
     M_thetadot = 0;
     N_alpha = .1803;
     N_delta = .0738;
     Tau_act = .02 ;         % Actuator first order time constant
     Velocity = 892.0;       % meters/sec


 % Performance Specifications

%      phi_rise = 5.0  ;   %   - removes rise time penalty
      phi_rise = .30  ;   % rise time specification


%       act_thresh = 1000.0 ;  %  - removes actuator response penalty
      act_thresh = 5.0 ;  % maximum actuator response specification
                          %  penalty threshold

%       hoe_thresh = 100.0 ;  %   - removes settling error penalty
      hoe_thresh = 0.0 ;  % settling error penalty threshold
                          % specification


  % Define time histories :

      dtl = .01 ;

      tl_final = 5.0 ;

      dts = .001 ;

      ts_final = 0.2 ;

      ts = 0 : dts : ts_final ;

      tl = ts_final : dtl : tl_final ;
```

```
% Define Airframe + First Order Actuator State Space Realization :

     A = [ - N_alpha  1.0  - N_delta
             M_alpha   0     M_delta
               0       0   - ( 1 / Tau_act ) ] ;

     B = [    0
              0
            ( 1 / Tau_act ) ] ;

     C = [ ( N_alpha * Velocity )  0   ( N_delta * Velocity ) ] ;


     D = [ 0 ] ;


% Determine LQR design for basic plant with first order actuator :

     qmatrix = [  10  0  0
                   0 .1  0
                   0  0 .01  ];

     rmatrix = .01;


% Formulate closed loop regulator with
%       linear state feedback control
%       to determine step response of closed loop system
% Set Cc & Dc matrices to force output equal to control for step
%       response

       hoec = 1.0 ;                % hang off error compensation

       Ac = A - B * kg ;

       Bc = hoec * B * kg ;

       Cc = - kg ;

       Dc =  kg ;


 % Determine the closed loop step response :

       Rc = [  1
               0
               0  ] ;

  % Small and large dt step responses :

       [ys,yl,xs,xl] = step2s(Ac,Bc,Cc,Dc,Rc,dts,ts,dtl,tl);

  % Evaluate the cost function separately for small and large dt
  %   responses and sum to get total :
```

```
      r = [ 1   0   0 ] ;     % indicates desired step response

      j1 = jcost( xs, r, ys, qmatrix, rmatrix, dts );

      j2 = jcost( xl, r, yl, qmatrix, rmatrix, dtl );

      j = j1 + j2 ;

% Evaluate the rise time

      lens = length( ts ) ;

      t_rise = risetime( xs(:,1), ts, .8 ) ;

      if t_rise >= ts( lens )
        t_rise = risetime( xl(:,1), tl, .8 ) ;
      end ;

% Determine infinity norm of actuator response

      act_norm = infnorm( xs(:,3) ) ;

% Determine the Hang Off Error in state 1 ( settling error ) :

      [ xrows, xcols ] = size( xl ) ;

      hang_off_error = xl( xrows, 1 ) - r( 1 ) ;

      if isnan( hang_off_error )
        hang_off_error = 10 ;
      end ;

      if abs( hang_off_error ) > 10
        hang_off_error = 10 ;
      end ;


% Convert cost function to suitably limited fitness function
%   including penality for exceeding rise time specification

      if isnan( j )
         j = 100 ;
      end ;

      if j > 100
        j = 100 ;
      end ;

      if t_rise > phi_rise
        cost_term = j  +  10.0 * ( t_rise - phi_rise ) ;
      else
        cost_term = j ;
      end ;
```

```
% Penalty for Settling Error

      if abs( hang_off_error ) > hoe_thresh
        cost_term = cost_term + 10.0 * abs( hang_off_error ) ;
      end ;

% Penalty for infinity norm of actuator response

      if abs( act_norm ) > act_thresh
        cost_term = cost_term ..
                        + 20.0 * abs( act_norm - act_thresh ) ;
      end ;

% Limit Cost Term

      if cost_term > 100
        cost_term = 100 ;
      elseif cost_term < .5
        cost_term = .5  ;
      end ;

      fitness1 = exp( 2 / cost_term ) - 1 ;

      fitness = - fitness1 ;          % Invert for use in FMINS

 %  End APFITFUN
```

B.2 CONV2NM2

This MATLAB function converts a 16 bit character string to an appropriately scaled real number ranging between the specified lower and upper limits.

A program listing of this function follows.

```
%  FUNCTION CONV2NM2
%  This function converts the input string variable ( actually
%    an 16 column matrix of zero's and one's ) to a number ( num )
%    with value between lower and upper.
%  Note that in some cases there may not be an exact zero value
%    after the conversion.

function [num] = conv2nm2( string, lower, upper )

d = [ 32768
      16384
       8192
       4096
       2048
       1024
        512
        256
        128
         64
         32
         16
          8
          4
          2
          1 ] ;

num = lower + ( upper - lower ) * ( string * d ) / 65535 ;

%  End of function CONV2NM2
```

B.3 CONV2STR

This MATLAB function converts a real number to a 16 bit character string, assuming it ranges between the specified lower and upper limits.

A program listing of this function follows.

```
%  FUNCTION CONV2STR
%  This function converts the input number to a 16 bit string
%    ( actually a 16 column matrix of zero's and one's ).
%  It is assummed the input number lies between the lower and
%   upper limit.  Usually the input number has been previously
%   generated from a string and scaled by using conv2nm2.

function [string] = conv2str( num, lower, upper )

rem = 65535 * ( num - lower ) / ( upper - lower ) ;

div = 32768 ;

string = zeros( 1, 16 ) ;

for i = 1:16

  bit = fix( rem / div ) ;

  string( i ) = bit ;

  rem = rem - bit * div ;

  div = div / 2 ;

end ;

%  End of function CONV2STR
```

B.4 EVALPOP7

This MATLAB function evaluates the genetic algorithm fitness function for a given system configuration and design specification. It converts the parameter string information into appropriately scaled gain values, and evaluates the fitness function for every member of the population. It first calculates the system step response, using STEP2S, and then determines the linear quadratic performance index using JCOST. It then evaluates the cost function penalties imposed by peak actuator response, settling error, and rise time constraints. The final cost function is limited and converted to a fitness function appropriate for the genetic algorithm.

A program listing of this function follows.

```
%    FUNCTION  EVALPOP7  - Evaluate Population function M file
%              - assuming three parameters per string : k1,k2,k3
%              - using LQR cost function
%              - includes penality for rise time specification
%              - use step2 function
%              - include actuator model
%              - include hang off error penalty
%              - include actuator max response penalty
%              - use smaller dt to calculate actuator response norm
%
%                by    R. Hull,    3/1/93
%
%

function [ pop_fitness, pop_param ] = evalpop7( A, B, C, D, ts, dts, ..
    tl, dtl, hoe_thresh, phi_rise, act_thresh, popsize, kg_scale, ..
     pop_chrom );

pop_param = [ ] ;

pop_fitness = [ ] ;

kg = [ 1.0  1.0  1.0 ] ;

% Set cost function weighting matrices

     qmatrix = [  10  0  0
                   0 .1  0
                   0  0 .01  ];
```

```
    rmatrix = .01;

% Population loop

    for k = 1:popsize

      kg(1) = conv2nm2( pop_chrom(k, 1:16), kg_scale(1,1),..
                                            kg_scale(1,2) ) ;

      kg(2) = conv2nm2( pop_chrom(k, 17:32), kg_scale(2,1),..
                                             kg_scale(2,2) ) ;

      kg(3) = conv2nm2( pop_chrom(k, 33:48), kg_scale(3,1),..
                                             kg_scale(3,2) ) ;

      pop_param = [ pop_param ; kg ] ;

% Formulate closed loop regulator with
%       linear state feedback control
%       to determine step response of closed loop system
% Set Cc & Dc matrices to force output equal to control

      hoec = 1.0 ;              % hang off error compensation

      Ac = A - B * kg ;

      Bc = hoec * B * kg ;

      Cc = - kg ;

      Dc =  kg ;

% Determine the closed loop step response :

      Rc = [  1
              0
              0  ] ;

 % Small and large dt step responses :

      [ys,yl,xs,xl] = step2s(Ac,Bc,Cc,Dc,Rc,dts,ts,dtl,tl);

 % Evaluate the cost function separately for small and large dt
 %   responses and sum to get total :

      r = [ 1  0  0 ] ;    % indicates desired step response

      j1 = jcost( xs, r, ys, qmatrix, rmatrix, dts );

      j2 = jcost( xl, r, yl, qmatrix, rmatrix, dtl );

      j = j1 + j2 ;

 % Evaluate the rise time
```

```
      lens = length( ts ) ;

      t_rise = risetime( xs(:,1), ts, .8 ) ;

      if t_rise >= ts( lens )
        t_rise = risetime( xl(:,1), tl, .8 ) ;
      end ;

% Determine infinity norm of actuator response

      act_norm = infnorm( xs(:,3) ) ;

% Determine the Hang Off Error in state 1 ( settling error ) :

      [ xrows, xcols ] = size( xl ) ;

      hang_off_error = xl( xrows, 1 ) - r( 1 ) ;

      if isnan( hang_off_error )
        hang_off_error = 10 ;
      end ;

      if abs( hang_off_error ) > 10
        hang_off_error = 10 ;
      end ;


% Convert cost function to suitably limited fitness function
%   including penality for exceeding rise time specification

      if isnan( j )
         j = 100 ;
      end ;

      if j > 100
        j = 100 ;
      end ;

      if t_rise > phi_rise
        cost_term = j  +  10.0 * ( t_rise - phi_rise ) ;
      else
        cost_term = j ;
      end ;

% Penalty for Settling Error

      if abs( hang_off_error ) > hoe_thresh
        cost_term = cost_term + 10.0 * abs( hang_off_error ) ;
      end ;

% Penalty for infinity norm of actuator response

      if abs( act_norm ) > act_thresh
        cost_term = cost_term ..
                      + 20.0 * abs( act_norm - act_thresh ) ;
```

```
      end ;

% Limit Cost Term

      if cost_term > 100
        cost_term = 100 ;
      elseif cost_term < .5
        cost_term = .5  ;
      end ;

      fitness = exp( 2 / cost_term ) - 1 ;

      pop_fitness = [ pop_fitness ; fitness ] ;

      % Write data to screen if not in continuous plot mode

      if cont_plot ~= 1

        k___fitness = [ k kg fitness ]

      end ;

      end     % for k loop

% End of EVALPOP7
```

B.5 EVALPOPT

This MATLAB function evaluates the genetic algorithm fitness function for the simple single parameter example used in program GATEST. It is only used with GATEST.

A program listing of this function follows.

```
%  FUNCTION  EVALPOPT  - Evaluate Population function M file
%               - assuming one parameter per string :  k1
%               - special function for GATEST program
%
%                 by    R. Hull,    3/22/93

function [ pop_fitness, pop_param ] = ..
                      evalpopt( popsize, kg_scale, pop_chrom );

pop_param = [ ] ;
pop_fitness = [ ] ;
kg = [ 1.0 ] ;

  for k = 1:popsize       % Population loop

    kg(1) = conv2nm2( pop_chrom(k, 1:16), kg_scale(1),..
                                          kg_scale(2) ) ;
    pop_param = [ pop_param ; kg ] ;

 %  Compute the special fitness function of one parameter :

    x1 = kg( 1 ) ;

    fitness =  -( x1-2.5 )^ 2 + 10.0 + 5*sin( 4*pi*x1 ) ;

    pop_fitness = [ pop_fitness ; fitness ] ;

    % Write data to screen if not in continuous plot mode

    if cont_plot ~= 1, k___fitness = [ k kg fitness ], end;

  end     % for k loop

 % End of EVALPOPT
```

B.6 INFNORM

This MATLAB function computes the peak response value, also known as the infinity norm, of the given input signal.

A program listing of this function follows.

```
%  FUNCTION INFNORM
%  This function computes the Infinity Norm of a
%   vector function input y.
%
%   where :
%      y = input response vector
%      norm = output scaler equal to the infinity norm
%               of y.
%
%     written in MATLAB by R. Hull  2/18/93

function  norm = infnorm( y ) ;

   abs_y = abs( y ) ;          % a vector

   norm = max( abs_y ) ;       % a scaler

 %  End of function infnorm
```

B.7 JCOST

This MATLAB function computes the weighted linear quadratic performance index, given the time history of the system state and control variables.

A program listing of this function follows.

```
%  FUNCTION JCOST
%  This function computes the Linear Quadratic Requlator
%   cost function based on matrices containing the
%   time history of the input and states of the system
%   where :
%              l = number of time samples
%              n = number of states
%              m = number of control inputs
%      x = history of the states    ( l x n )
%      r = desired step response of states ( 1 x n )
%      u = history of the controls  ( l x m )
%      Q = state weighting matrix   ( n x n )
%      R = control weighting matrix ( m x m )
%      dt = time sample interval ( constant )

%  written in MATLAB by R. Hull  10/14/92

function [j] = jcost(x,r,u,Q,R,dt)

   j = 0 ;
   jdotp = 0 ;

   for i = 1:length(x(:,1)) ;

     xerr = r - x(i,:) ;

     jdot = xerr * Q * xerr.' + u(i,:) * R * u(i,:).' ;

     j = j + .5 * dt * ( jdot + jdotp ) ;

     jdotp = jdot ;

     end;
 % End of Function JCOST
```

B.8 RISETIME

This MATLAB function computes risetime, given a response history, as measured from 0 to 80 % of the commanded value.

A program listing of this function follows.

```
%  FUNCTION RISETIME
%  This function computes the Rise Time of a step
%   response function to the value input as y_value.
%   If the step response never reaches the desired
%   y_value, the rtime returned will be the final
%   value in the time vector.
%
%   where :
%      y = step response vector
%      t = time vector
%      y_value = desired value of y at which response time
%                is measured
%      r_time = earliest time for which step response in y
%               always exceeds y_value.  Note that no
%               interpolation is performed.
%
%     written in MATLAB by R. Hull  1/25/93

function  rtime = risetime( y, t, y_value ) ;

   y_index = length( y ) ;

   for i = 1 : length( y )
      if y(i) >= y_value
         y_index = i ;
         break ;
       end ;
   end ;

   len_t = length( t ) ;

   if y_index < len_t
     rtime = t( y_index ) ;
   else
     rtime = t( len_t )  ;
   end ;

 %  End of function RISETIME
```

B.9 SELECT

This MATLAB function selects two parents from the available population pool for mating as part of the genetic algorithm. The probability of selection is based on a uniformly distributed random function, but is weighted according to each individual's fitness relative to the total population fitness. It ensures that the two parents selected are different individuals.

A program listing of this function follows.

```
%   FUNCTION  SELECT  - Select Parents for Genetic Algorithm
%
%                 by    R. Hull       1/12/93
%
%    Note : this selection procedure insures that
%           parent_1 and parent_2 are different individuals.

function  [ parent_1, parent_2 ] = select( popsize, sum_fitness, ..
                                                    pop_fitness ) ;

%  Determine random roulette wheel pointer

pointer_sum = rand * sum_fitness ;

part_sum = 0 ;

for parent_1 = 1 : popsize

  part_sum = part_sum + pop_fitness( parent_1 ) ;

  if part_sum >= pointer_sum, break, end ;   % break loop

end ;   %   ( for parent_1 loop )

% Remove parent_1 from choices and repeat to find parent_2

sum_fitness = sum_fitness - pop_fitness( parent_1 ) ;

pop_fitness( parent_1 ) = 0 ;

%  Determine second random roulette wheel pointer

pointer_sum = rand * sum_fitness ;

part_sum = 0 ;
```

```
for parent_2 = 1 : popsize
  part_sum = part_sum + pop_fitness( parent_2 ) ;
  if part_sum >= pointer_sum, break, end ;   % break loop
end ;   %   ( for parent_2 loop )
%  End of function SELECT
```

B.10 SELXSITE

This MATLAB function selects the crossover site for the crossover function of the genetic algorithm. It implements an experimental crossover function, in which the random distribution of the site selection is weighted in favor of high order bits in early generations, gradually shifting to favor low order bits in later generations.

A program listing of this function follows.

```
%   FUNCTION  SELXSITE  - Select Genetic Algorithm Crossover Site
%
%                  by    R. Hull       2/15/93
%
%          This function selects a crossover site for the
%          Genetic Algorithm that is stochastically weighted
%          depending upon generation.
%          It uses a gaussian distribution that is "folded over"
%          at the ends of the parameter bit length.

function  xsite = selxsite( paramlength, generation ) ;

% Switch to Normal Distribution

rand('normal') ;

sigma = paramlength / 3 ;

mean = generation ;

if mean < 1
  mean = 1 ;
elseif mean > paramlength
  mean = paramlength ;
end ;

gauss_max = paramlength - 1 ;

% Generate Gaussian number and fold over negative tail

gauss_num = abs( rand * sigma + mean ) ;

% Fold over positive side tail

if gauss_num > ( 2 * gauss_max )
  gauss_num = 2 * gauss_max ;
end ;
```

```
if gauss_num > gauss_max
  gauss_num = 2 * gauss_max - gauss_num ;
end ;

% Convert to integer crossover site between 1 and paramlength

xsite = fix( gauss_num ) + 1 ;

% Switch Back to Uniform Distribution

rand('uniform') ;

%  End of function SELXSITE
```

B.11 STEP2S

This MATLAB function computes the step response to a LTIC system using two different integration time intervals. It first uses a small dt (dts) to provide accurate integration during the initial period of high dynamics. It then switches to a larger dt (dtl) to compute the steady state step response during a period of lower dynamic rates. This enables an accurate response to be generated, yet conserves computation time and resources.

A program listing of this function follows.

```
%       Function STEP2S
%       Computes the step response of continuous-time linear
%         systems, using split dt's.
%
%       This function  calculates the response of the system:
%
%         x = Ax + Bu
%         y = Cx + Du
%
%      to a step input vector R ( m x 1 ), using dts over
%       the time history in ts, and dtl over the time history
%       in tl.
%
%          ys is the response over time history ts
%          yl is the response over time history tl
%          xs is the state history over time history ts
%          xl is the state history over time history tl
%
%       This allows an accurate step response to be determined over
%       the initial period of high actuator dynamics using a small
%       dt (dts), then a steady state response to be generated using
%       a large dt (dtl), without using too much computer time.
%       The final states of the small dt response become the initial
%       states for the large dt response.
%
% by     R. Hull        3/1/93

function [ys,yl,xs,xl] = step2s(a,b,c,d,r,dts,ts,dtl,tl) ;

% Convert Continuous System to Discrete Time Systems :

 [aas,bbs] = c2d(a,b,dts);

 [aal,bbl] = c2d(a,b,dtl);
```

```
% Build Small Step Input Matrix from input vector
 lens = length(ts);
 u = ones( lens, 1 ) * r.' ;
% Compute linear time invariant state response :
 xs = ltitr(aas,bbs,u);
% Compute output vector :
 ys = xs * c.'+ u * d.';
% Build Large Step Input Matrix from input vector
 lenl = length(tl);
 u = ones( lenl, 1 ) * r.' ;
% Initialize states to final values from small dt response :
 xsf = xs( lens, : ) ;
% Compute linear time invariant state response :
 xl = ltitr(aal,bbl,u,xsf);
% Compute output vector :
 yl = xl * c.'+ u * d.';
% End of function STEP2S
```